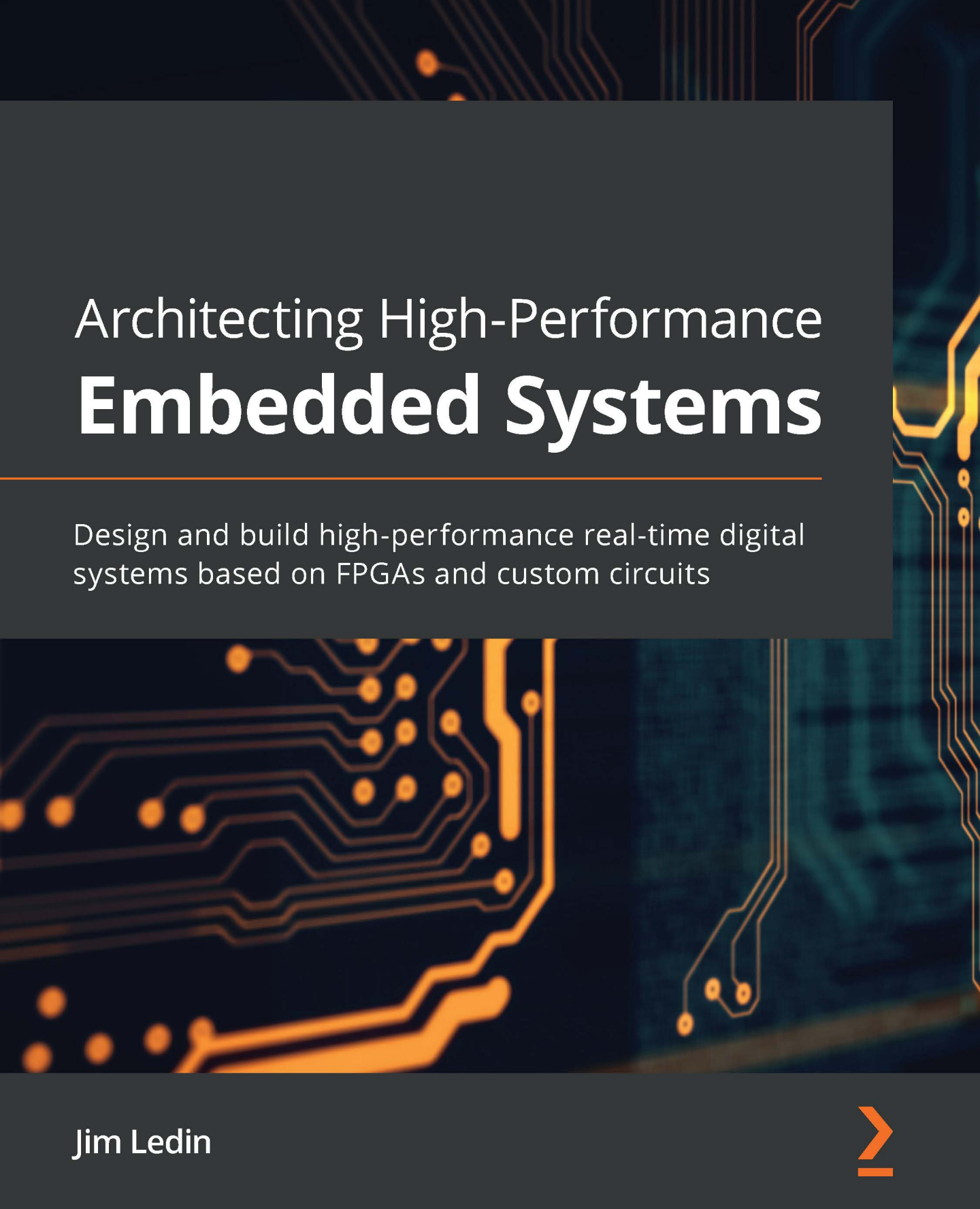




Architecting High-Performance **Embedded Systems**

Design and build high-performance real-time digital systems based on FPGAs and custom circuits



Architecting High-Performance Embedded Systems

Design and build high-performance real-time digital systems based on FPGAs and custom circuits

Jim Ledin



BIRMINGHAM—MUMBAI

Architecting High-Performance Embedded Systems

Copyright © 2021 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Group Product Manager: Wilson D'souza

Associate Publishing Product Manager: Sankalp Khattri

Senior Editor: Rahul Dsouza

Content Development Editor: Nihar Kapadia

Technical Editor: Nithik Cheruvakodan

Copy Editor: Safis Editing

Project Coordinator: Neil D'mello

Proofreader: Safis Editing

Indexer: Priyanka Dhadke

Production Designer: Joshua Misquitta

First published: January 2021

Production reference: 1060121

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-78995-596-5

www.packt.com



Packt . com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at [packt . com](http://packt.com) and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at [customercare@packtpub . com](mailto:customercare@packtpub.com) for more details.

At [www . packt . com](http://www.packt.com), you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Contributors

About the author

Jim Ledin is the CEO of Ledin Engineering, Inc. Jim is an expert in embedded software and hardware design, development, and testing. He is also accomplished in embedded system cybersecurity assessment and penetration testing. He has a B.S. degree in aerospace engineering from Iowa State University and an M.S. degree in electrical and computer engineering from the Georgia Institute of Technology. Jim is a registered professional electrical engineer in California, a **Certified Information System Security Professional (CISSP)**, a **Certified Ethical Hacker (CEH)**, and a **Certified Penetration Tester (CPT)**.

I would like to thank Mike Anderson for his thoughtful technical review of this book. I would also like to thank Nihar Kapadia and all of the staff at Packt for their competent assistance in completing this book during a very challenging year.

About the reviewer

Mike Anderson is currently a senior project lead and embedded systems architect for The Aerospace Corporation, based in Chantilly, VA. With over 40 years in the embedded and real-time computing industry, Mike works with a number of **Real-Time Operating System (RTOS)** offerings for **Internet of Things (IoT)** devices. However, his focus over the past decade is primarily embedded Linux on a number of CPU architectures. Mike is a regular speaker at the Embedded Linux Conference, Sensors Expo, and other Linux-, embedded systems-, and IoT-oriented conferences. His ongoing projects include work with mesh wireless topologies, AI/ML on satellites, 6LoWPAN, and working as a mentor for multiple high school robotics with the FIRST Robotics Program.

I'd like to thank my family for their infinite patience with my taking the time to review this book. I feel that it's time well spent for the future of the industry.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Table of Contents

Preface

Section 1: Fundamentals of High-Performance Embedded Systems

1

Architecting High-Performance Embedded Systems

Technical requirements	4	Real-time operating systems	14
Elements of embedded systems	4	FPGAs in embedded systems	16
Power source	5	Digital logic gates	16
Digital processing	6	Flip-flops	19
Software and firmware	7	Elements of FPGAs	20
Specialized circuitry	8	FPGA synthesis	22
Input from the environment	8	Hardware design languages	22
Output to the environment	9	The benefits of using FPGAs in embedded system designs	25
Operating in real time	11	Xilinx FPGAs and development tools	26
Periodic operation	12		
Event-driven operation	13	Summary	27

2

Sensing the World

Technical requirements	30	The types of sensors used in embedded systems	36
Introducing passive, active, and smart sensors	30	Light	36
Applying analog-to-digital converters	32	Temperature	37
		Pressure	37

Humidity	38	GPS	43
Fluid flow	38		
Force	39	Communicating with sensors	44
Ultrasonic	39	GPIO	44
Audio	39	Analog voltage	48
Magnetic	40	I2C	50
Chemical	40	SPI	51
Ionizing radiation	41	CAN bus	52
Radar	41	Wireless	54
Lidar	41		
Video and infrared	42	Processing sensor data	55
Inertial	43	Summary	56

3

Operating in Real Time

Technical requirements	58	Dynamic memory allocation	74
What does real-time mean?	58	Deadlock	76
Attributes of a real-time embedded system	59	Priority inversion	77
Performing multiple tasks	60	Popular real-time operating systems	81
Rate-monotonic scheduling	69	embOS	83
		FreeRTOS	83
Understanding key RTOS features and challenges	70	INTEGRITY	84
Mutexes	70	Neutrino	84
Semaphores	72	µc/OS-III	84
Queues	72	VxWorks	85
Event flags	73	Summary	86
Timers	73		

Section 2:
Designing and Constructing High-Performance Embedded Systems

4

Developing Your First FPGA Program

Technical requirements	90	Testing the implementation	105
Using FPGAs in real-time embedded system designs	90	Developing your first FPGA project	106
Block RAM and distributed RAM	91	Project description	106
FPGA I/O pins and associated features	92	Installing the Vivado tools	107
Specialized hardware resources	94	Creating a project	110
Processor cores	95	Creating VHDL source files	112
FPGA implementation languages	95	Testing the logic behavior	120
VHDL	95	Defining I/O signals	127
Verilog	97	Creating a top-level VHDL file	129
Block diagrams	98	Synthesizing and implementing the FPGA bitstream	130
C/C++	99	Downloading the bitstream to the board	133
The FPGA development process	100	Programming the bitstream to onboard flash memory	135
Defining system requirements	101	Summary	139
Allocating functionality to the FPGA	102		
Identifying required FPGA features	102		
Implementing the FPGA design	103		

5

Implementing systems with FPGAs

Technical requirements	142	Algorithms using nonstandard data sizes	163
The FPGA compilation process	142	Kicking off the oscilloscope FPGA project	164
Design entry	142	Project description	164
Logic synthesis	147	Baseline Vivado project	165
Design optimization	149	Summary	176
High-level synthesis	153		
Optimization and constraints	160		
Algorithm types most suitable for FPGA implementation	162		
Algorithms that process high-speed data streams	162		
Parallel algorithms	163		

6

Designing Circuits with KiCad

Technical requirements	178	Adding signal labels	197
Introducing KiCad	178	Adding global labels	197
Basic KiCad procedures	180	Creating differential signal pairs	198
Placing and connecting circuit components	182	Creating offboard connections	198
Creating component symbols	190	Symbol annotation and electrical rules checking	199
Developing the project schematic diagram	195	Laying out the PCB	200
Adding text annotations	196	Prototyping the circuit board	210
		Summary	211

7

Building High-Performance Digital Circuits

Technical requirements	214	Reflow soldering	227
Circuit board assembly tools and procedures	214	Preparing for assembly and placing parts	230
Optical magnification	214	Reflow soldering and hand soldering	233
Tweezers	216	Hand soldering	234
Flux	217	Post-assembly board cleaning and inspection	236
Solder	218	Summary	239
Electrostatic discharge protection	219		
Hand soldering	220		
Solder wick	222		
Solder paste application	223		

Section 3: Implementing and Testing Real-Time Firmware

8

Bringing Up the Board for the First Time

Technical requirements	244	Supplying power to the board	244
Preparing for power-up	244		

Checking our circuit's basic functionality	246	Cutting PCB traces	261
Testing the board power supplies	247	Installing solder jumpers and jumper wires	262
Testing the analog amplifiers	250	Removing components	263
Testing the ADC	253	Adding components	264
Adapting the circuit in case of problems	261	Generating the ADC encoder clock and 1 KHz calibration signal	265
		Summary	270

9

The Firmware Development Process

Technical requirements	272	Indicate that constant things are constant	296
Designing and implementing the FPGA algorithm	272	Automated code formatters	297
Digital oscilloscope system overview	272	Statically analyzing source code	297
Adding the deserializer	275	What is static code analysis?	297
Adding a FIFO buffer	280	Static code analysis tools	298
Adding the AXI bus interface	283	Using static code analysis effectively	299
Adding the MQTT protocol	285	Working with existing code	299
Coding style	291	Begin with only the most severe messages	301
Naming things	291	Resolving analyzer output messages	302
Comments in code	292	Common source code analyzer messages	302
Avoid literal numeric values	292	Source code version control	304
Braces, indentation, and vertical spacing	292	Version control with Git	304
Prioritize readability and correctness	293	Test-driven development	304
Avoid premature optimization	294	TDD applied to embedded systems	305
Avoid implementation-defined behavior	294	Summary	306
Avoid unconditional jumps	295		
Minimize the scope of identifiers	295		

10

Testing and Debugging the Embedded System

Technical requirements	308	Designing system-level tests	308
-------------------------------	------------	-------------------------------------	------------

Requirements-driven testing	309	attempt to duplicate it	332
Testing under nominal and off-nominal conditions	312	Determine whether the input is correct	333
Unit testing versus functional testing	313	Find ways to gain visibility into the system	333
Negative testing and penetration testing	315	Using a binary search debugging process	335
Testing in a simulated environment	316	temporarily removing portions of functionality	336
Achieving repeatable test results	316	Make the smallest program that demonstrates the problem	336
Developing a test plan	317		
Conducting tests and recording results	319	Summary of best practices for high-performance embedded system development	337
Identify the data to be collected	319	Designing for test	337
Configuring the system under test	320	Leave room for growth	338
Executing the test procedures	320	Design hardware with future capabilities in mind	338
Quick-look assessment of test results	321	Developing only the code you need right now	339
Repeating tests when necessary	322	Maintain strict version control	340
Regression testing existing code	322	Develop unit tests as code is in development	341
Ensuring comprehensive test coverage	323	Begin system-level testing as soon as functionality is present	341
Requirements traceability matrix	324		
Tracking code coverage	327	Summary	342
Dealing with syntax and compilation errors and warnings	330		
Working with static code analysis and unit testing	331		
Define the problem clearly and			

Other Books You May Enjoy

Index

Preface

Modern digital devices used in homes, in cars, and on our persons contain increasingly sophisticated computing capabilities. These embedded systems generate, receive, and process digital data streams at rates of up to multiple gigabits per second. This book teaches you how to use **field-programmable gate arrays (FPGAs)** and high-speed digital circuit design techniques to create your own cutting-edge digital device designs.

Intended audience for this book

This book is intended for software developers, hardware engineers, **Internet of Things (IoT)** developers, and anyone else seeking to understand the process of developing high-performance embedded systems. The potential audience includes anyone with an interest in learning about the fundamentals of FPGA development and all aspects of firmware development in C and C++. Readers should have a basic level of familiarity with the C language, digital circuits, and soldering electronic components.

What this book covers

Chapter 1, Architecting High-Performance Embedded Systems, introduces the elements of embedded system architectures and discusses some key system features that are common across a wide variety of embedded applications. An embedded system generally includes at least one microcontroller or microprocessor, sensors, actuators, a power source, and, in many cases, one or more network interfaces. The chapter continues with an exploration of the relationship between embedded systems and the IoT.

Chapter 2, Sensing the World, introduces the principles and implementations of sensors used in a wide variety of embedded applications. Passive sensors measure attributes of the world such as temperature, pressure, humidity, light intensity, and atmospheric composition. Active sensors use energy-emitting technologies such as radar and lidar to detect objects and measure their position and velocity.

Chapter 3, Operating in Real Time, addresses the need for embedded systems to generate real-time responses to inputs measured from sensors and other sources. The concepts of **Real-Time Operating Systems (RTOSes)** and their key features are introduced, as well as some challenges that commonly occur when implementing multitasking in real-time applications. The chapter concludes with a presentation of the important characteristics of some popular open source and commercial RTOS implementations.

Chapter 4, Developing Your First FPGA Program, begins with a discussion on the effective use of FPGA devices in real-time embedded systems and continues with a description of the functional elements contained within standard FPGAs. The range of FPGA design languages, including **Hardware Description Languages (HDLs)**, block diagram methods, and popular software programming languages including C and C++, is introduced. The chapter continues with an overview of the FPGA development process and concludes with a complete example of an FPGA development cycle, starting with a statement of system requirements and ending with a functional system implemented in a low-cost FPGA development board.

Chapter 5, Implementing Systems with FPGAs, dives into the process of designing and implementing embedded devices with FPGAs. It begins with a description of the FPGA compilation software tools that convert a description of a logic design in a programming language into an executable FPGA configuration. We will discuss the types of algorithms best suited to FPGA implementation and suggest a decision-making approach for determining whether a particular embedded system algorithm is more appropriately implemented using a traditional processor or with an FPGA. The chapter ends with the step-by-step development of a baseline FPGA-based processor project that will be expanded to implement a high-speed digital oscilloscope using circuitry and software developed in later chapters.

Chapter 6, Designing Circuits with KiCad, introduces the excellent open source KiCad electronics design and automation suite. Working in KiCad, you will design a circuit using schematic diagrams and develop a corresponding printed circuit board layout. You'll learn how to turn a circuit board design into a prototype at a very reasonable cost. This chapter includes example schematics for the oscilloscope circuit project you will assemble in the next chapter.

Chapter 7, Building High-Performance Digital Circuits, presents the processes and techniques involved in assembling prototype high-performance digital circuits using surface-mount and through-hole electronic components. A recommended set of tools is identified, including a soldering station, a magnifier or microscope, and tweezers for handling tiny parts. The reflow soldering process is introduced, along with descriptions of some low-cost options for implementing a small-scale reflow capability.

Chapter 8, Bringing Up the Board for the First Time, covers how, having designed, constructed, cleaned, and inspected the printed circuit board, it is now time to apply power – in other words, perform the infamous smoke test. This chapter leads you through the process of carefully providing first-time power to the board and checking basic circuit-level functionality. If you discover any problems, the chapter contains suggested approaches for addressing them. After passing these tests, it is time to add to the FPGA logic and test the digital interface to the oscilloscope board.

Chapter 9, The Firmware Development Process, shows how, now that we have a functioning circuit board, to flesh out the remaining key portions of the FPGA algorithm, including communication with the **Analog to Digital Converter (ADC)**, and continue development of the MicroBlaze processor firmware. When developing firmware, it is important to subject the code to static analysis where possible, which can head off many errors that are otherwise difficult to debug. It is also important to implement a version control system to track the evolution of the code over the project life cycle. We will discuss the importance of developing a comprehensive, at least partially automated test suite to maintain code quality as changes are made. The chapter recommends some free and commercial tools for performing each of these functions.

Chapter 10, Testing and Debugging the Embedded System, discusses how, as the development of our embedded system nears completion, the time arrives to conduct thorough testing in the context in which it will operate. This testing must address the entire expected range of environmental conditions and user inputs, including invalid inputs, to ensure proper operation under all conditions. The chapter concludes with a discussion of recommended debugging procedures and a summary of best practices for high-performance embedded system development.

To get the most out of this book

This book takes full advantage of powerful free commercial and open source software tool suites to develop FPGA algorithms and to design sophisticated printed circuit boards. To follow along with the example project, you will need a specific FPGA development board, the Digilent Arty A7-100. To construct the digital circuits to implement your designs, you will need a set of tools for soldering and desoldering surface mount components. You will also need tools to assist in working with fine-scale parts, such as precision tweezers and a magnifier or microscope.

Software/hardware covered in the book	OS Requirements
Xilinx Vivado	Windows and Linux
KiCad	Windows, macOS, and Linux
Arty A7-100	Windows and Linux

If you are using the digital version of this book, we advise you to access the code via the GitHub repository (link available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

The code bundle for the book is hosted on GitHub at <https://github.com/PacktPublishing/Architecting-High-Performance-Embedded-Systems>. In case there's an update to the code, it will be updated at this GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: http://www.packtpub.com/sites/default/files/downloads/9781789955965_ColorImages.pdf.

Conventions used

There are a number of text conventions used throughout this book.

`Code in text`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "The term `std_logic` refers to a single-bit binary data type."

A block of code is set as follows:

```
architecture BEHAVIORAL of FULL_ADDER is
begin
    S      <= (A XOR B) XOR C_IN;
    C_OUT <= (A AND B) OR ((A XOR B) AND C_IN);
end architecture BEHAVIORAL;
```

Any command-line input or output is written as follows:

```
dism /online /Enable-Feature /FeatureName:TelnetClient
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Leave the selections at their default values and click **Next**."

Tips or important notes

Appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at customercare@packtpub.com.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packtpub.com/support/errata, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packt.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit authors.packtpub.com.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit packt.com.

Section 1: Fundamentals of High-Performance Embedded Systems

This part introduces the basic concepts of embedded systems, real-time computing, and **Field Programmable Gate Array (FPGA)** devices. It provides a high-level overview of topics that will be covered in detail in later chapters.

This part of the book comprises the following chapters:

- *Chapter 1, Architecting High-Performance Embedded Systems*
- *Chapter 2, Sensing the World*
- *Chapter 3, Operating in Real Time*

1

Architecting High-Performance Embedded Systems

This chapter introduces the elements of embedded system architectures and discusses some key system features that are common across a wide variety of embedded applications. An embedded system generally includes at least one microcontroller or microprocessor, sensors, actuators, a power source, and, in many cases, one or more network interfaces. The chapter continues with an exploration of the relationship between embedded systems and the **Internet of Things (IoT)**.

This chapter emphasizes the necessity for many types of embedded systems to function in a real-time manner and presents the basic embedded system operating sequence of reading from input devices, computing outputs, and updating output devices in a repetitive manner while remaining synchronized with the passage of time.

The chapter concludes with an introduction to digital logic and the **Field-Programmable Gate Array (FPGA)**, and identifies the design space within the spectrum of embedded systems most appropriately addressed by these high-performance devices.

After completing this chapter, you will have a broad understanding of the components that make up embedded systems and the relationship of embedded systems to the IoT. You will know why many embedded systems must operate in synchronization with real time and will understand the basic structure of FPGAs and how they can be employed to implement high-performance embedded systems.

We will cover the following topics in this chapter:

- Elements of embedded systems
- The Internet of Things
- Operating in real time
- FPGAs in embedded systems

Technical requirements

The files for this chapter are available at <https://github.com/PacktPublishing/Architecting-High-Performance-Embedded-Systems>.

Elements of embedded systems

Embedded systems are everywhere. Almost any electrical device you interact with that is more complicated than a simple light switch contains a digital processor that reads input data from its environment, executes a computational algorithm, and generates some kind of output that interacts with the environment.

From the moment you open your eyes in the morning (in response to an alarm produced by a digital device), to brushing your teeth (with an electric toothbrush that contains a digital processor), to toasting a breakfast bagel (in a digitally controlled toaster oven), to disabling your (digital) home alarm system, you interact with embedded devices. Throughout the day, you provide input to, and receive output from, many other devices, such as television remote controls, traffic signals, and railroad crossings. Highly digitized transportation systems, including automobiles, airplanes, and passenger ferries, each contain dozens, if not hundreds, of embedded processors that manage drive train operation, oversee safety features, maintain a comfortable climate, and provide entertainment for the humans they carry.

Let's take a moment to clarify the sometimes-murky dividing line separating embedded systems from general-purpose computing devices. The attribute that defines an embedded computing system is the integration of digital processing within a device that has some larger purpose beyond mere computing. Devices that do not contain any type of digital processing are not embedded systems. For example, an electric toothbrush that contains only a battery and a motor controlled by an on-off switch is not an embedded system. A toothbrush containing a microcontroller that illuminates a red light when you press down too hard while brushing is an embedded system.

A desktop computer, even though it is capable of performing many tasks, and can be enhanced through the addition of a wide variety of peripherals, is just a computer. An automobile, on the other hand, has as its primary purpose the transportation of passengers. In performing this function, it relies on a variety of subsystems containing embedded processing. Automobiles are embedded systems. Personal computers are not.

A smartphone is more difficult to clearly categorize in terms of membership in the set of embedded systems. When in use as a telephone, it is clearly performing a function consistent with the definition of an embedded system. When using it as a web browser, though, it more closely resembles a small general-purpose computer. Clearly, it is not always possible to definitively determine whether a device is an embedded system.

It is helpful to understand differences in the operating environment of general-purpose computers in comparison to embedded devices. Personal computers and enterprise servers tend to work best in climate-controlled indoor settings. Embedded devices such as those in automobiles are often exposed to far more rugged conditions, including the full effects of rain, snow, wind, dust, and heat.

A large percentage of embedded devices lack any sort of active cooling system (which is standard in personal computers and server computers) and must ensure their internal components remain at safe operating temperatures regardless of external conditions.

Embedded systems, whether they are relatively simple devices or highly complex systems, are typically composed of the following elements.

Power source

All electronic digital devices require some a of power. Most commonly, embedded systems are powered by utility electrical power, batteries, or by the host system in which the device operates. For example, an automobile taillight assembly containing a processor and a CAN bus communication interface is powered by 12 volts **Direct Current (DC)** provided by the car's electrical system.

It is also possible to power embedded devices from rechargeable batteries connected to solar panels that allow the device to continue operation at nighttime and on cloudy days, or even by harvesting energy from the environment. A self-winding wristwatch uses energy harvested from arm motion to generate mechanical or electrical power. Safety- and security-critical embedded systems often use utility power as the primary power source while also providing batteries as backup power to enable operation during power outages.

Time base

Embedded systems generally require some means of tracking the progress of time, also known as **wall clock time**, both in the short term (for durations of microseconds and milliseconds) and in the long term, keeping track of the date and time of day. Most commonly, a primary system clock signal is generated using a crystal oscillator or a **Microelectromechanical System (MEMS)** oscillator that produces an output frequency of a few megahertz.

A crystal oscillator amplifies the resonant vibration of a physical crystal, typically made of quartz, to generate a square wave electrical signal using the piezoelectric effect. A MEMS oscillator contains a vibrating mechanical structure that produces an electrical output using electrostatic transduction.

Once set to the correct time, a clock driven by a crystal oscillator or a MEMS oscillator will exhibit small errors in frequency (typically 1-100 parts per million) that accumulate over periods of days and weeks to gradually drift by seconds and then minutes away from the correct time. To mitigate this problem, most internet-connected embedded devices periodically access a time server to reset their internal clocks to the current time.

Digital processing

Embedded computing systems, by definition, contain some form of digital processor. The processing function is generally provided by a microcontroller, a microprocessor, or a **system on a chip (SoC)**. A **microcontroller** is a highly integrated device that contains one or more **central processing units (CPUs)**, **random access memory (RAM)**, **read-only memory (ROM)**, and a variety of peripheral devices. A **microprocessor** contains one or more CPUs, but has less of the overall system functionality integrated in the same device in comparison to a microcontroller, typically relying on external circuits for RAM, ROM, and peripheral interfaces.

An SoC is even more highly integrated than a microcontroller, generally combining one or more microcontrollers with additional digital hardware resources configured to perform specialized functions at high speed. As we will see in the *FPGAs in embedded systems* section and in subsequent chapters, SoC designs can be implemented as FPGA devices in architectures combining traditional microcontrollers with custom, high-performance digital logic.

Memory

Embedded systems generally contain RAM for working memory as well as some type of ROM, often flash memory, to store executable program code and other required information such as static databases. The quantity of each type of memory must be sufficient to meet the needs of the embedded system architecture over its planned life cycle. If the device is intended to support firmware upgrades, sufficient memory resources must be provided in the hardware design to support the anticipated range of potential system capability enhancements over its lifetime.

Software and firmware

In traditional computing environments, the executable code that users work with, such as web browsers and email programs, is referred to as **software**. This term is used to differentiate program code from the **hardware** that makes up the physical components of the computer system. In general-purpose computers, software is stored as files on some type of disk drive. In embedded systems, executable code is usually stored in some type of ROM, which is a hardware component within the device. Because of this arrangement, we can contemplate that the code occupies a middle ground between hardware and software. This middle ground is referred to as **firmware**. In the early days of embedded systems, code was often burned into a memory device that could not be changed after the initial programming. These devices were more hardware-like (hence more firm) than most currently produced embedded devices, which often contain rewriteable flash memory. Nevertheless, we continue to use the term *firmware* to describe code programmed into embedded systems.

Specialized circuitry

Embedded systems support a wide variety of applications, some of which are relatively simple processes such as monitoring button presses on a television remote control and producing the corresponding output signal, while other types of systems perform extremely complex processing-intensive work on high data rate input signals. While a simple embedded system may be able to use a tiny microcontroller to perform all of the digital processing required, a more complex system may require processing resources that exceed the capabilities of off-the-shelf microcontrollers and more capable microprocessors such as x86 and ARM processors.

In years past, architects of these more sophisticated embedded designs would turn to an **application-specific integrated circuit (ASIC)** to implement custom circuitry to perform the processing at the speed needed for proper system operation. An ASIC is an integrated circuit containing a custom digital circuit designed to support a particular application. The production of ASIC devices typically involves a very expensive production setup phase, which makes their use impractical during project prototyping and for small production runs.

Fortunately, much of the capability afforded by ASICs is available in low-cost FPGA devices. Because FPGAs are easily reprogrammable, they are generally used for embedded system prototyping and in low volume production runs. For high-volume production (thousands or millions of units), the lower per-unit cost of an ASIC can make the production setup costs worthwhile. This book will focus on the use of FPGAs in the prototyping of embedded systems.

Input from the environment

Embedded systems generally require input from their environment, whether it is from a human operating a user interface or from sensors measuring certain aspects of the system or environment in which they operate. For example, a battery-electric vehicle powertrain controller will track various aspects of the vehicle state, such as battery voltage, motor current, vehicle speed, and the position of the accelerator pedal. The system architecture must include hardware peripherals to measure input from each of the sensors with the necessary precision. The overall powertrain control system must be capable of performing measurements from all sensors at the rate required for proper vehicle operation.

Output to the environment

In addition to reading inputs from the environment, the embedded system will generally produce one or more outputs for use by human operators or by the host system. Continuing the battery-electric vehicle example, the powertrain controller uses the accelerator pedal position, along with other inputs, to compute a command to the controller for the drive motor. This command adjusts the torque output of the drivetrain.

In addition to directly supporting system operation, embedded controllers often provide output for human consumption, such as displaying the vehicle speed in the dashboard. Each output must be updated at a rate sufficient to support proper system operation, including the needs of human perception. When implementing human interfaces, graphical outputs should update smoothly without visible glitches or flicker and audio outputs must avoid timing-related problems such as gaps or skips.

Network communication

While many simple embedded systems operate in a completely self-contained manner, reading their inputs, computing outputs, and updating output devices in an isolated context, more and more embedded system designs support some form of network communication. This capability enables device features such as remote notifications from home video doorbells and the continuous monitoring of machinery on factory floors.

Enhancing an embedded system with an always available network communication capability can provide significant enhancements to functionality. However, this feature also presents a security risk that may be exploited by malicious actors if developers aren't careful to emphasize security within the system architecture. It is important to understand and address the security risks introduced by the inclusion of communication capabilities in an embedded system architecture.

Embedded system architects combine these elements to produce a system design that performs its intended functions, with appropriate safety margins, across the entire range of anticipated environmental operating conditions.

A suitable system design satisfies additional requirements such as size and weight constraints and power consumption limits, and holds production costs to an acceptable level. The design constraints for an embedded system depend heavily on such attributes as the number of units that will be produced, safety-critical aspects of the system, and the need for operation in rugged conditions.

There may be additional considerations that surface during the selection of the microcontroller or microprocessor architectural family and associated tools, such as the availability of suitable programming language compilers and debuggers. The selection of a processor family may depend in part on the past experience of the development team. It also depends on the cost, availability, and anticipated learning curve associated with the development tools.

Embedded system architectures that include persistent communication capability must address an additional dimension of the design space involving communications between individual devices and centralized nodes (typically servers accessed over the internet) and interactions between users and the embedded systems.

The widespread deployment of small-scale embedded systems with network connectivity has introduced the term **Internet of Things (IoT)**. The next section discusses the relevance of IoT to the architectures of embedded systems.

The Internet of Things

Conceptually, the IoT represents an effort to maximize the utility of large numbers of disparate embedded devices through massive network communication. The feature that distinguishes IoT devices from more mundane embedded systems is the presence of a communication path between each device and one or more central nodes that gather data from the sea of devices and, in many cases, allow authorized users to issue commands to individual devices and to collections of devices.

During the IoT device development process, particularly when developing devices that will have access to sensitive personal information (such as home security cameras), responsible embedded system architects must undertake extensive measures to ensure the security of the end devices. IoT devices are often installed in consumer's homes, and security breakdowns that allow malicious actors to take control of cameras, microphones, or security systems must be prevented to the maximum extent possible. Although the system designer cannot prevent every security mistake an end user might commit, a more secure system can assist the user by taking steps such as guiding the selection of strong passwords and by being resistant to common types of attacks such as brute force password guessing.

Examples of IoT devices and systems include the following:

- *A home alarm system consisting of window and door sensors and motion sensors:* This type of system generally includes a smartphone app providing immediate notification of alarm events. This system not only notifies the alarm company to initiate a response to alarm events, it also notifies the homeowner to the occurrence of those events. Clearly, this type of alarm system must be resistant to cyberattacks that would render the alarm function ineffective.

- *Electrical lights and power outlets:* Many different illumination devices are available with internet-based monitoring and control, including light bulbs, light fixtures, and power strips capable of switching lights on and off. The app associated with each of these devices allows remote control of individual lights as well as the scheduling of light turn-on and turn-off times throughout the day. As with IoT alarm systems, security is an important feature that must be fully integrated into the system design.
- *Smart speakers:* IoT speakers such as *Amazon Echo* and *Google Nest* provide a voice interface that allows users to make requests in natural language. Users preface commands with a word or phrase to wake up the speaker, such as "Alexa" or "Hey Google," followed by a command or request. These devices enable interaction with a variety of other IoT devices, including alarm systems and lighting control. An example voice command is "Alexa, turn on the lights."
- *Medical monitoring and treatment:* A wide variety of embedded devices is deployed in hospitals and home environments to monitor aspects of patient health, such as temperature, blood oxygen, heart rate, breathing, and many more. These devices often communicate with a centralized database to enable tracking of current and historical health patterns by medical professionals. Other digital systems perform active treatment functions, such as infusing medications, and assisting with breathing.
- *Industrial applications:* Embedded systems are widely used in factory lines, energy generation systems, energy transmission systems, and in the oil and gas industries to monitor and control complex systems and processes. For example, a broad range of sensors and actuators is required to perform real-time monitoring and management of the operation of an oil pipeline that may be thousands of miles long.

This book is focused on the architecture and design of embedded systems. We will examine all aspects of the design of IoT embedded systems, including network communication. We will discuss IoT security requirements for embedded systems as well as the communication protocols used to monitor and control IoT embedded devices.

Embedded devices usually operate under tight time constraints. The next section introduces the key aspects of real-time operation and the approaches embedded systems use to synchronize with the passage of time.

Operating in real time

To satisfy an embedded system's real-time requirements, the system must sense the state of its environment, compute a response, and output that response within a prescribed time interval. These timing constraints generally take two forms: periodic operation and event-driven operation.

Periodic operation

Embedded systems that perform periodic updates intend to remain in synchronization with the passage of time in the real world over long periods of time. These systems maintain an internal clock and use the passage of time as measured by the system clock to trigger the execution of each processing cycle. Most commonly, processing cycles repeat at fixed time intervals. Embedded systems typically perform processing at rates ranging from 10 to 1,000 updates per second, though particular applications may update at rates outside this range. *Figure 1.1* shows the processing cycle of a simple periodically updated embedded system:

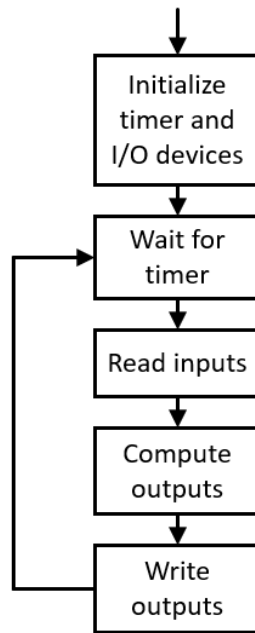


Figure 1.1 – Periodically updated embedded system

In the system of *Figure 1.1*, processing starts at the upper box, where initialization is performed for the processor itself and for the **input/output (I/O)** devices used by the system. The initialization process includes configuring a timer that triggers an event, typically an **interrupt**, at regularly spaced points in time. In the second box from the top, processing pauses while waiting for the timer to generate the next event. Depending on the capabilities of the processor, waiting may take the form of an idle loop that polls a timer output signal, or the system may enter a low power state waiting for the timer interrupt to wake the processor.

After the timer event occurs, the next step, in the third box from the top, consists of reading the current state of the inputs to the device. In the following box, the processor performs the computational algorithm and produces the values the device will write to the output peripherals. Output to the peripherals takes place in the final box at the bottom of the diagram. After the outputs have been written, processing returns to wait for the next timer event, forming an infinite loop.

Event-driven operation

Embedded systems that respond to discrete events may spend the vast majority of their time in an idle state and only come to life when an input is received, at which time the system executes an algorithm to process the input data, generates output, writes the output to a peripheral device, and then goes back to the idle state. A pushbutton-operated television remote control is a good example of an event-driven embedded device. *Figure 1.2* shows the processing steps for an event-driven embedded device:

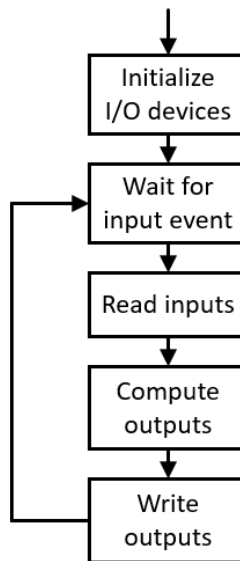


Figure 1.2 – Event-driven embedded system

Most of the processing steps in an event-driven embedded system are similar to those of the periodic system, except the initiation of each pass through the computational algorithm is triggered by an input to the device. Each time an input event occurs, the system reads the input device that triggered the event, along with any other inputs that are needed. The processor computes the outputs, writes outputs to the appropriate devices, and returns to wait for the next event, again forming an infinite loop. The system may have inputs for many different events, such as presses and releases of each of the keys on a keypad.

Many embedded systems must support both periodic and event-driven behaviors. An automobile is one example. While driving, the drivetrain processors sense inputs, perform computations, and update outputs to manage the vehicle speed, steering, and braking at regular time intervals. In addition to these periodic operations, the system contains other input signals and sensors that indicate the occurrence of events, such as shifting into gear or the involvement of the vehicle in a collision.

For a small, microcontroller-based embedded system, the developer might write the entirety of the code, including all timing-related functions, input, and output via peripheral interfaces, and the algorithms needed to compute outputs given the inputs. Implementing the blocks of *Figure 1.1* or *Figure 1.2* for a small system might consist of a few hundred lines of C code or assembly language.

At the higher end of system complexity, where the processor might need to update various outputs at different rates and respond to a variety of event-type input signals, it becomes necessary to segment the code between the time-related activities, such as scheduling cyclic updates, and the code that performs the computational algorithms of the system. This segmentation becomes particularly critical in highly complex systems that contain hundreds of thousands or even millions of lines of code. Real-time operating systems provide this capability.

Real-time operating systems

When a system architecture is of sufficient complexity that the separation of time-related functionality from computational algorithms becomes beneficial, it is common to implement an operating system to manage lower-level functionality, such as scheduling time-based updates and managing responses to interrupt-driven events. This allows application developers to focus on the algorithms required by the system design, which includes their integration into the capabilities provided by the operating system.

An operating system is a multilayer suite of software providing an environment in which applications perform useful functions, such as managing the operation of a car engine. These applications execute algorithms consisting of processor instruction sequences and perform I/O interactions with the peripheral devices needed to complete their tasks.

Operating systems can be broadly categorized into real-time and general-purpose operating systems. A **real-time operating system (RTOS)** provides features to ensure that responses to inputs occur within a specified time limit, as long as some assumptions about how the application code behaves remain true. Real-time applications that perform tasks such as managing the operation of a car engine or a kitchen appliance typically run under an RTOS to ensure that the electrical and mechanical components they control receive responses to any change in inputs within a specified time.

Embedded systems often perform multiple functions simultaneously. The automobile is a good example, where one or more processors continuously monitor and control the operation of the powertrain, receive input from the driver, manage the climate control, and operate the sound system. One method of handling this diversity of tasks is to assign a separate processor to perform each function. This makes the development and testing of the software associated with each function straightforward, though a possible downside is that the design ends up with a plethora of processors, many of which don't have very much work to do.

Alternatively, a system architect may assign more than one of these functions to a single processor. If the functions assigned to the processor perform updates at the same rate, integration in this manner may be straightforward, particularly if the functions do not need to interact with each other.

In the case where multiple functions that execute at different rates are combined in the same processor, the complexity of the integration will increase, particularly if the functions must transfer data among themselves.

In the context of an RTOS, separate periodically scheduled functions that execute in a logically simultaneous manner are called tasks. A **task** is a block of code with an independent flow of execution that is scheduled in a periodic or event-driven manner by the operating system. Some operating systems use the term thread to represent a concept similar to a task. A **thread** is a flow of code execution, while the term *task* generally describes a thread of execution combined with other system resources required by the task.

Modern RTOS implementations support the implementation of an arbitrary number of tasks, each of which may execute at different update rates and at different priorities. The **priority** of an RTOS task determines when it is allowed to execute relative to other tasks that may be simultaneously ready to execute. Higher-priority tasks get the first chance to execute when the operating system is making scheduling decisions.

An RTOS may be *preemptive*, meaning it has the authority to pause the execution of a lower-priority task when a higher-priority task becomes ready to run. When this happens, which typically occurs when it becomes time for the higher-priority task to perform its next update, or when a blocked I/O operation initiated by the higher-priority task completes, the system saves the state of the lower-priority task and transfers control to the higher-priority task. After the higher-priority task finishes and returns to the waiting state, the system switches back to the lower-priority task and resumes its execution.

As we'll see in later chapters, there are several additional features available in popular RTOS implementations such as FreeRTOS. There are also some significant performance constraints that developers of applications running in an RTOS environment must be aware of to avoid problems such as higher-priority tasks entirely blocking the execution of lower-priority tasks, and the possibility of deadlock between communicating tasks.

In the next section, we will introduce the basics of digital logic and examine the capabilities of modern FPGA devices.

FPGAs in embedded systems

A **gate array** is a digital integrated circuit containing a large number of logic elements that can be connected in an arbitrary manner to form complex digital devices. Many FPGAs even support the implementation of a full-blown microcontroller together with an array of I/O devices. A microcontroller or microprocessor implemented using the gates of an FPGA is referred to as a **soft processor**.

Early versions of gate arrays were one-time programmable devices in which a circuit design would be implemented within a device at the factory where the device was constructed, or perhaps by system developers using a programming device connected to their desktop computers. Once a device had been programmed, it could not be changed. Since that time, the technology of gate arrays has improved and now reprogrammable gate arrays are widely available.

Today, there is a tremendous variety of **Field-Programmable Gate Arrays (FPGAs)** available even to system developers of modest means. As the name implies, FPGAs are gate arrays that can be reprogrammed at any time, even after an embedded system has been assembled and delivered to its end user.

Before we get into the specifics of FPGA devices, we'll introduce some underlying concepts related to digital circuits, specifically logic gates and flip-flops.

Digital logic gates

A modern FPGA device contains what we might think of as a large box of digital parts that can be used to assemble complex logic circuits. The simplest of these components include the **AND**, **OR**, and **XOR** gates that perform basic logic functions. Each of these gates has two inputs and one output. The **NOT** gate is even simpler, with one input and one output. Logic gates operate on the binary input values 0 and 1 and produce an output of 0 or 1 as determined by the inputs.

In reality, the binary values in these circuits are represented by a voltage, with 0 usually represented as a low voltage (near zero volts) and 1 as a higher voltage that depends on the technology of the circuitry in which the gates are implemented. A common level for the 1 value in modern circuitry is 3.3 volts.

We will briefly discuss the behavior of each of these gates and present the gate's schematic symbol and the truth table that defines the gate's behavior. The behavior of a logic gate can be represented as a **truth table** where, for each possible combination of inputs, the output is given. Each column represents one input or output signal, with the output shown at the right side of the table. Each row presents one set of input values with the output of the gate given those inputs.

The AND gate outputs a 1 when both of its inputs are 1, otherwise the output is 0. *Figure 1.3* is the AND gate schematic symbol:



Figure 1.3 – AND gate schematic symbol

The following table is the truth table for the AND gate:

A	B	Output
0	0	0
1	0	0
0	1	0
1	1	1

The OR gate outputs a 1 if either of its inputs is 1, otherwise the output is 0. *Figure 1.4* is the OR gate schematic symbol:

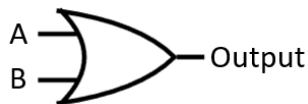


Figure 1.4 – OR gate schematic symbol

The following table is the truth table for the OR gate:

A	B	Output
0	0	0
1	0	1
0	1	1
1	1	1

The XOR gate outputs a 1 if exactly one of its outputs is 1, otherwise the output is 0. *Figure 1.5* is the XOR gate schematic symbol:



Figure 1.5 – XOR gate schematic symbol

The following table is the truth table for the XOR gate:

A	B	Output
0	0	0
1	0	1
0	1	1
1	1	0

The NOT gate has a single input and an output that is the inverse of its input: An input of 0 produces an output of 1, and an input of 1 produces an output of 0. *Figure 1.6* is the NOT gate schematic symbol:

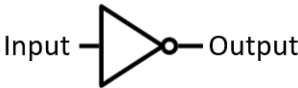


Figure 1.6 – NOT gate schematic symbol

In *Figure 1.6*, the triangle represents an amplifier, meaning this is a device that turns a weaker input signal into a stronger output signal. The circle represents the inversion operation.

The following table is the truth table for the NOT gate:

Input	Output
0	1
1	0

Each of the AND, OR, and XOR gates can be implemented with an inverting output. The function of an inverting gate is the same as described, except the output is the opposite of the output from the non-inverting gate. The schematic symbol for an AND, OR, or XOR gate with inverted output has a small circle added at the output side of the symbol, just as on the output of the NOT gate. The names of the gates with inverted outputs are **NAND**, **NOR**, and **XNOR**. The letter *N* in each of these names indicates NOT. For example, NAND means NOT AND, which is functionally equivalent to an AND gate followed by a NOT gate.

Flip-flops

A device that changes its output state only when a clock signal makes a specified transition (either low-to-high or high-to-low) is referred to as an **edge-sensitive device**. A **flip-flop** is an edge-sensitive device that holds one bit of data as its output signal. The flip-flop updates the data value it contains based on the state of its input signal when the clock input receives the specified transition.

The *positive edge-triggered D flip-flop* is a common digital circuit component that finds use in a variety of applications. The D flip-flop typically includes **set** and **reset** input signals that force the stored value to 1 (set) or to 0 (reset). This type of flip-flop has a data input called the **D** input.

The D flip-flop has a clock input that triggers the transfer of the **D** input to the **Q** output on the clock's rising edge. The $\bar{Q}$ output (the overbar here means NOT) always has the opposite binary value from the **Q** output. Other than within an extremely narrow window of time surrounding the rising edge of the clock signal, the flip-flop does not respond to the value of the **D** input. When active (at the 1 level), the **S** (set) and **R** (reset) inputs override any activity on the **D** and clock inputs.

Figure 1.7 shows the schematic symbol for the D flip-flop. The clock input is indicated by the small triangle on the left side of the symbol:

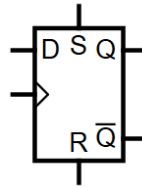


Figure 1.7 – D flip-flop

The truth table for the D flip flop is shown below. The upward-pointing arrows in the CLK column indicate the rising edge of the clock signal. The Q and $\bar{Q}$ outputs on the table rows containing upward-pointing arrows in the CLK column represent the state of the outputs following the rising clock edge. In this table, the value X indicates *don't care*, meaning it does not matter what value that signal has in determining the Q output. The output Qprev prev represents the most recent value of Q produced through the action of the S, R, D, and CLK inputs:

S	R	D	CLK	Q	$\bar{Q}$
0	0	1	↑	1	0
0	0	0	↑	0	1
0	0	X	Stable	Q_{prev}	$\bar{Q}_{\text{prev}}$
1	0	X	X	1	0
0	1	X	X	0	1

Any digital circuit composed of a collection of logic gates is referred to as **combinational logic** when the output at any moment depends only on the current state of the inputs. In other words, the output does not depend on previous input values. Combinational logic circuits have no memory of past inputs or outputs.

Armed with this background information on logic gates and flip-flops, we will next discuss the implementation of circuits composed of these and related components in FPGAs.

Elements of FPGAs

The digital parts available within an FPGA typically fall into the categories of lookup tables, flip-flops, block RAM, and DSP slices. We will briefly examine each of these components.

Lookup tables

Lookup tables are used extensively in FPGAs to implement combinational logic circuits constructed from simple logic gates such as NOT, AND, OR, and XOR, as well as the siblings of the last three of these with inverted outputs: NAND, NOR, and XNOR.

Rather than implementing a logic gate circuit in hardware with the actual gates in its design, it is always possible to represent the same circuit using a simple lookup table. Given any combination of input signals, the correct output can be retrieved from a memory circuit addressed by the inputs. A typical FPGA lookup table has six single-bit input signals and a single bit output. This is equivalent to a single-bit-wide memory device with six address inputs holding 64 bits of data ($2^6 = 64$). Circuits that require fewer than six inputs can treat some of the inputs as *don't care* inputs. Circuits with greater complexity can combine multiple lookup tables to produce their results.

Flip-flops

For a digital circuit to retain any record of past events, some form of memory is required. As presented in the previous section, a flip-flop is a high-speed single-bit memory storage device. As with lookup tables, FPGAs contain large numbers of flip-flops to support the construction of complex sequential logic circuits. Digital circuitry that generates outputs based on a combination of current inputs and past inputs is called **sequential logic**. This is in contrast to combinational logic, where outputs depend only on the current state of the inputs.

Block RAM

A **Block RAM (BRAM)** is a range of dedicated memory locations within an FPGA. In comparison to traditional processor hardware, flip-flops can be likened to processor registers, while BRAM is more like cache memory. **Cache memory** in a processor is used to temporarily store copies of recently accessed memory contents in a memory area where the processor can access it again, if it needs to, much faster than reaching out to main memory. FPGA synthesis tools allocate BRAM to circuit designs in a manner that optimizes the performance of the digital circuit.

DSP slices

A **DSP slice** is a section of digital logic optimized to perform the central computation of digital signal processing – the **Multiply-Accumulate (MAC)** operation. MAC processing involves multiplying two lists of numbers element by element and adding the products together. As a simple example, if two sequences are defined as a_0, a_1, a_2 and b_0, b_1, b_2 , the result of a MAC operation on these sequences is $a_0b_0 + a_1b_1 + a_2b_2$. Many DSP algorithms are built upon repetitive MAC operations performed with a list of algorithm-specific coefficients on a stream of input data.

Other functional elements

Every FPGA manufacturer expends significant effort to ensure each FPGA model provides the highest performance possible for use in a wide range of application areas. In order to better meet a diversity of needs, FPGAs often include hardware implementations of additional categories of low-level digital components such as shift registers, carry logic, and multiplexers. The inclusion of these hardware elements enables the synthesis of better-performing algorithms in comparison to an FPGA that generates these low-level components from the more generic resources available within the device.

The next section introduces the FPGA synthesis process, which converts a high-level description of an FPGA algorithm into a circuit implementation within a specific FPGA device.

FPGA synthesis

Although an FPGA device contains a large collection of low-level digital building blocks used to implement complex digital devices, it is important for system developers who are new to FPGA technology to understand that, in most cases, designers do not need to work directly at the level of these components. Instead, digital designers specify the system configuration as a combination of higher-level predefined functional blocks, such as a soft processor, and custom digital logic defined using a hardware description language. It is also possible to specify FPGA algorithms using programming languages such as C and C++.

The process of converting the high-level description of device functionality into the allocation and interconnection of the lookup tables, flip-flops, BRAM, and other device components is called **FPGA synthesis**. The synthesis process is conceptually similar to the software compilation process that converts human-readable source code to a binary program that can be executed by a processor.

Hardware design languages

It is easy to represent simple digital circuits using logic diagrams based on the schematic symbols presented earlier in this chapter. When designing digital devices that are very complex, however, the use of logic diagrams quickly becomes unwieldy. As an alternative to the logic diagram, a number of hardware description languages have been developed over the years.

The two most popular hardware design languages are VHDL and Verilog. **VHDL** is a multilevel acronym where the **V** stands for **VHSIC**, which means **Very High-Speed Integrated Circuit**, and **VHDL** stands for **VHSIC Hardware Description Language**. The syntax and some of the semantics of VHDL are based on the Ada programming language. **Verilog** has capabilities similar to VHDL. Although the two languages are not equivalent, it is broadly true that almost any digital design that you might implement in one of these languages can be implemented in the other language.

To provide a quick comparison between schematic diagram-based logic design and designing with a hardware description language, we will look at a simple adder circuit. A **full adder** adds two data bits plus an incoming carry bit and produces a one-bit sum and a carry output bit. This circuit, shown in *Figure 1.8*, is called a full adder because it includes the incoming carry in the calculation. A **half adder**, in comparison, adds only the two data bits without an incoming carry:

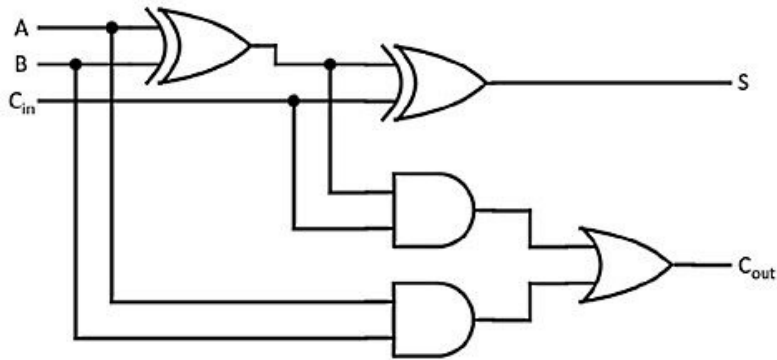


Figure 1.8 – Full adder circuit

The full adder uses logic gates to produce its output as follows: The sum bit S is 1 only if the total number of 1 bits in the collection A , B , C_{in} is an odd number. Otherwise, S is 0. The two XOR gates perform this logical operation. C_{out} is 1 if both A and B are 1, or if just one of A and B is 1 and C_{in} is also 1. Otherwise, C_{out} is 0.

The VHDL code in the following listing defines a digital circuit that performs the equivalent full adder function:

```
-- Load the standard libraries

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
```

```
-- Define the full adder inputs and outputs

entity FULL_ADDER is
  port (
    A      : in    std_logic;
    B      : in    std_logic;
    C_IN   : in    std_logic;
    S      : out   std_logic;
    C_OUT  : out   std_logic
  );
end entity FULL_ADDER;

-- Define the behavior of the full adder

architecture BEHAVIORAL of FULL_ADDER is

begin

  S      <= (A XOR B) XOR C_IN;
  C_OUT <= (A AND B) OR ((A XOR B) AND C_IN);

end architecture BEHAVIORAL;
```

This code is a fairly straightforward textual description of the full adder in *Figure 1.8*. Here, the section introduced with `entity FULL_ADDER is` defines the input and output signals of the full adder component. The `architecture` section toward the end of the code describes how the circuit logic operates to produce the outputs `S` and `C_OUT` given the inputs `A`, `B`, and `C_IN`. The term `std_logic` refers to a single-bit binary data type. The `<=` characters represent wire-like connections that drive the output on the left-hand side with the value computed on the right-hand side.

It is important, especially for FPGA developers coming from a software background, to understand that there is no concept of sequential execution in VHDL code. The statements in the `BEHAVIORAL` section at the end of the code that associate the outputs `S` and `C_OUT` with logical expressions are defining a digital circuit equivalent to *Figure 1.8*. They are *not* specifying computations that execute in sequence as in a traditional software program.

The benefits of using FPGAs in embedded system designs

For embedded system architects who are new to developing with FPGAs, the many benefits of using these devices may not be immediately obvious. Although FPGAs certainly are not appropriate for every embedded system design, it is useful to consider whether the use of FPGA technology is appropriate for your next system design.

Some of the benefits of developing embedded systems with FPGAs are as follows:

- **Processor customization:** Because the soft processors used in FPGAs are programmed into the device, it is standard for the developers of these products to provide a variety of configuration alternatives to the end user. Some common options are a choice between a 64-bit or 32-bit processor, the inclusion or exclusion of a floating-point processor, and the inclusion or exclusion of instructions that require significant hardware resources, such as integer division. These are just a few of the options that are likely to be available. The soft processor configuration can be modified even late in the development cycle to optimize trade-offs between system performance and FPGA resource utilization.
- **Flexible peripheral configuration:** Since the I/O interfaces in an FPGA design are defined in software, designers can include exactly the I/O devices they need and avoid including I/O hardware they don't need. As with processor customization, it is straightforward to modify the types and the number of I/O devices even late in the development cycle.
- **High-level synthesis:** Modern FPGA development tools support the definition of computationally intensive algorithms in traditional programming languages, including C and C++. This allows system developers with a software skill set to develop algorithms in a traditional software development environment and directly transition the same code into an optimized FPGA implementation. The FPGA version of the algorithm is relieved of traditional processor-based restrictions, such as sequential instruction execution and a fixed memory architecture. The high-level synthesis tools will generate an FPGA implementation that exploits execution parallelization and defines a memory architecture best suited to the algorithm. A custom hardware algorithm can be combined with a soft processor to implement a complete, high-performance digital system on a single FPGA device.

- **Hardware acceleration for parallelizable applications:** Any algorithm that benefits from parallelization is a candidate for implementation as custom FPGA logic. Rather than executing an algorithm sequentially with processor instructions, FPGA hardware can often perform the processing in parallel much faster. Many modern FPGA devices contain dedicated hardware to support **digital signal processing (DSP)** operations. These capabilities are available for use by many types of parallel algorithms, such as digital filtering and neural networks.
- **Extensive debugging capabilities:** Soft processors often provide options to enable a variety of debugging capabilities, such as instruction tracing, multiple complex breakpoints, and the ability to monitor the innermost operations of the processor and its interactions with other system components at the hardware level. As system development wraps up, developers can remove resource-intensive debugging capabilities from the final design to enable deployment in a smaller and less costly FPGA device.
- **Rapid prototyping of ASIC designs:** For embedded system designs intended to support the high volume that makes ASIC usage cost-effective, it is helpful to perform early prototyping with FPGAs to validate the system's digital design prior to investing in an ASIC implementation. The use of FPGAs in this context enables rapid development iterations that enable extensive testing of the new features introduced at each build iteration.

Xilinx FPGAs and development tools

There are several manufacturers of FPGA devices and the development tools associated with them. To avoid trying to cover multiple vendors and their FPGA devices and development toolchains, and to avoid discussing these topics at too abstract a level, we are going to select one vendor and one set of development tools for use in the examples and projects developed in this book. This is not to suggest that another vendor's devices and tools aren't as good, or possibly better, for the applications we will discuss. We are simply choosing to use Xilinx FPGA devices and development tools to make the steps we are taking concrete and to allow you to follow along.

The *Vivado Design Suite* is available as a free download from Xilinx, though you will need to create a Xilinx user account to access the download page. Visit <https://www.xilinx.com/> and select the option to create your account. Once you are logged in on the website, visit <https://www.xilinx.com/support/download.html> and download the Vivado Design Suite.

Vivado can be installed on Windows and Linux operating systems. Our projects in the coming chapters can be developed with Vivado running under either operating system.

Vivado includes a set of simulation capabilities that will allow you to develop and execute FPGA implementations within the simulation environment at zero cost. When you decide you need to see your FPGA design run on an actual FPGA, the best option for the projects we will be covering is the Arty A7-100T. This board currently costs US\$249 and is available at <https://store.digilentinc.com/arti-a7-artix-7-fpga-development-board-for-makers-and-hobbyists/>.

Summary

This chapter introduced the elements of embedded systems, including digital processing, sensors, actuators, a power source, and, in many cases, one or more communication interfaces. We continued with an exploration of the relationship between embedded systems and the IoT.

The necessity for embedded systems to function in a real-time manner was emphasized, and the basic operational sequence of reading from input devices, computing outputs, and updating output devices was presented.

This chapter introduced the FPGA in functional terms and identified the benefits these high-performance devices bring to embedded system designs.

Having completed this chapter, you should now have a broad understanding of the components that make up embedded systems and the relationship between embedded systems and the IoT. You should also know why and how embedded systems operate in real time and understand how FPGAs can be employed to implement high-performance embedded systems.

In the next chapter, we will look at the range of sensors that are commonly used to enable embedded systems to receive input from users and from the environment around them.

2

Sensing the World

This chapter introduces the principles and applications of sensors used in a wide variety of embedded systems. Passive sensors measure attributes of the world such as temperature, pressure, humidity, light intensity, and atmospheric composition. Active sensors use energy-emitting technologies such as radar and lidar to detect objects and measure their position and velocity.

We will examine a broad range of sensor types as well as the communication protocols used for transferring sensor data into a processor. The chapter also discusses the processing an embedded system must perform on raw sensor measurements to provide actionable data for use by a processing algorithm.

After completing this chapter, you will have learned about many of the different types of sensors used in embedded systems and will understand what passive and active sensors are and will be familiar with several types of passive and active sensors. You will also understand some of the data processing methods commonly used on the raw measurements provided by sensors.

We will cover the following topics in this chapter:

- Introducing passive, active, and smart sensors
- Applying analog-to-digital converters
- The types of sensors used in embedded systems
- Communicating with sensors
- Processing sensor data

Technical requirements

The files for this chapter are available at <https://github.com/PacktPublishing/Architecting-High-Performance-Embedded-Systems>.

Introducing passive, active, and smart sensors

As we discussed in *Chapter 1, Architecting High-Performance Embedded Systems*, the basic sequence of processing in a simple embedded system consists of reading inputs, computing outputs, writing outputs, and waiting until either it is time to start the next processing loop or the next triggering event occurs. This chapter will look in more depth at the first of these steps: reading inputs. The inputs used by a particular system depend, obviously, on what the system does. In embedded systems, the inputs generally consist of commands entered by a user, commands received from other sources such as a network server controlling the system, and sensor measurements. Our focus here is on inputs collected using sensors.

In the context of embedded systems, a **sensor** is an electrical or electronic component that is sensitive to some property of its environment and produces an output corresponding to the measured property. To make this abstract description a bit more concrete, consider the operation of a thermistor to measure temperature. A **thermistor** is an electrical resistor that varies in resistance in a predictable manner as its temperature changes. By using a circuit to measure the resistance of the thermistor, an embedded system can estimate the temperature at the thermistor's location.

A thermistor is an example of a **passive sensor**. Passive sensors measure aspects of the environment, such as temperature or light intensity, by responding directly to the measured parameter. Passive sensors do not do anything to perturb the environment as they make their measurements.

An **active sensor**, on the other hand, generates some kind of stimulus that it applies to the environment. This type of sensor produces a measurement based on the response to the stimulus.

An ultrasonic distance sensor is one type of active sensor. As depicted in *Figure 2.1*, this sensor measures the distance to nearby objects by transmitting a pulse of acoustic energy at an ultrasonic frequency, and then senses any echo that comes back in response to the pulse:

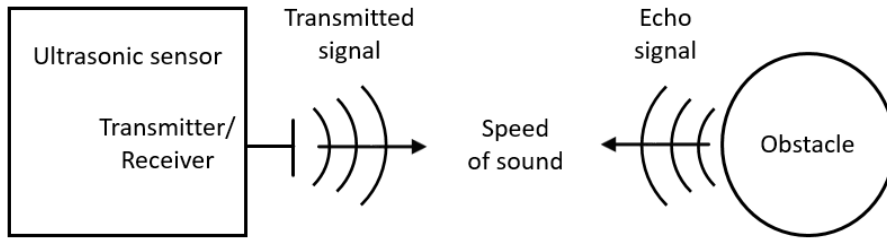


Figure 2.1 – Ultrasonic distance sensor

During each measurement, the sensor waits long enough for the pulse, traveling at the speed of sound, to reach its maximum expected measurement range and return to the sensor after bouncing off any physical object the pulse encounters. By measuring the time between the transmission of the pulse and the receipt of the echo, also referred to as the time of flight, the embedded system can determine the distance to the object. Other examples of active sensors include radar and lidar sensors used in modern automotive applications.

Some types of sensors are capable of operating in both passive and active modes. For example, sonar systems used underwater can passively measure the sounds of an ocean environment, including those produced by living creatures, by man-made systems, and by natural processes. Some sonar systems can also operate in an active mode, in which the system generates a *ping* and listens for echoes from objects encountered by the acoustic wave in the same manner as an ultrasonic sensor.

A simple analog sensor such as a thermistor requires several additional circuit components to enable measurement of resistance changes. Once the resistance has been measured (which is determined indirectly by measuring a voltage), the embedded processor must perform a computation to convert the voltage reading to the corresponding temperature.

A **smart sensor** offloads some of the circuit complexity and computational effort by integrating the measurement function and the conversion to engineering units into a single device, often a small module. Smart sensors generally contain a microcontroller and employ a digital interface to communicate the measured value to the host processor. Some smart sensors even allow developers to customize the onboard microcontroller code.

The use of smart sensors in an embedded system design can substantially simplify the hardware design. In particular, sensitive circuitry, like the amplifiers used to boost faint input signals, can come self-contained in a smart sensor. This avoids the difficulty associated with designing dedicated sensor circuitry for a system, though the cost of a smart sensor may be greater than the cost of the components an equivalent custom circuit design would require. The decision to select a smart sensor in particular applications will depend on factors such as sensor cost, anticipated production volume, and time to market pressure.

The next section presents some basic circuit configurations that interface common sensor types with embedded processors.

Applying analog-to-digital converters

Many types of sensors produce a response that can be measured as a voltage. An embedded processor measures a voltage with an analog-to-digital converter. An **analog-to-digital converter (ADC)** is a processor peripheral that samples an analog voltage and produces as output a digital data value corresponding to the voltage at the time of the sample.

An ADC is characterized by the number of bits in the digital measurement word, the voltage range of the input signal, the time it takes for a conversion to complete, and other performance parameters such as accuracy and measurement noise.

As shown in *Figure 2.2*, an analog voltage can vary continuously over time, and can take on any value within its operating range. The output of an ADC is only available at discrete points in time and can only take on the limited number of values dictated by its resolution. In this simplified example, the ADC produces measurements three bits wide with output values ranging from 0 to 7:

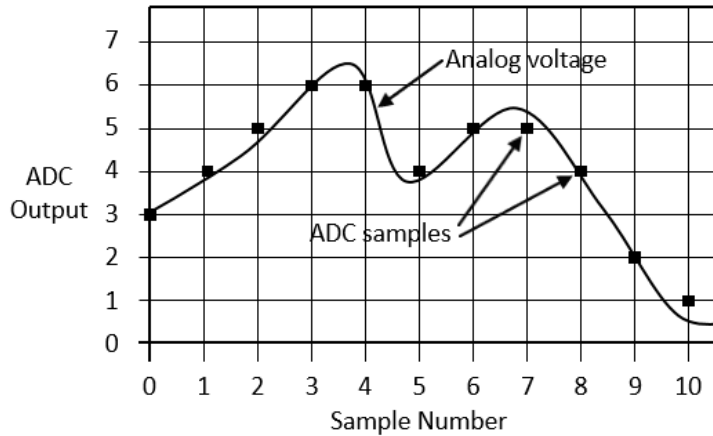


Figure 2.2 – Analog to digital conversion

ADCs are widely available with measurement bit widths from 8 to 18 bits. Although some extremely high-performance ADCs are available with sample rates of over one billion samples per second, in many embedded system applications, ADCs sample their inputs at a rate of ten times per second or even less.

Many low-cost microcontrollers integrate one or more ADCs within the processor circuit die. These devices allow the sampling of multiple analog input signals with resolutions of typically 10 or 12 bits at rates of up to hundreds of thousands of times per second.

Even with a very fast ADC, it takes some time to convert an analog input into a digital reading. To prevent changes in the analog input voltage during the measurement process from affecting the measurement, it is common to use a sample-and-hold circuit to freeze the analog voltage for the duration of the measurement process.

A **sample-and-hold** circuit is an analog circuit that passes its input voltage directly to its output whenever the *hold* input signal is inactive. When the *hold* input is active, the device freezes its output voltage at the voltage that was present when the *hold* input was activated. As part of an analog measurement circuit, the sample-and-hold component presents a constant voltage during each ADC conversion.

It is possible to connect multiple analog input signals in sequence to an ADC for measurement using analog multiplexing circuitry. An **analog multiplexer** has multiple analog inputs and, under the control of a set of digital input signals, can connect any of these inputs to its output. Microcontrollers and FPGA devices containing ADCs typically provide the option of using several I/O pins as analog inputs. When it is time to measure the voltage at any of these pins, the processing logic selects the appropriate analog multiplexer input channel and then measures the voltage on the corresponding input pin.

Analog signals generally contain corrupting influences referred to as **noise**. The noise produced in analog circuits comes from external sources and from within the embedded system itself. External noise sources include nearby electrical devices such as fluorescent lighting and household appliances producing electric fields that alter the voltages in the analog measurement circuitry. The primary internally generated source of noise in the analog signals in embedded systems is the digital circuitry present in the device. Each time a digital clock or gate changes state, the transition produces a tiny pulse that generates an electric field. The switching of logic gates also causes fluctuations in power supply voltages, which can affect analog readings. It is necessary for embedded system architects to take steps to reduce the influence of noise on analog measurements to an acceptable level.

The FPGA device on the Arty A7-100T board discussed in the *Xilinx FPGAs and development tools* section of *Chapter 1, Architecting High-Performance Embedded Systems*, contains an integrated 12-bit ADC module named XADC that is capable of performing measurements at up to 1 **million samples per second (MSPS)**. The XADC is a dual-channel device, which means it can measure two analog inputs simultaneously. The analog inputs are measured as the voltage differential between positive and negative input signals.

This device can be configured to operate in two input modes: unipolar and bipolar. In unipolar mode, the ADC input voltage range is 0 to 1V. An input of 0V produces a measured output (in hexadecimal) of 000. An input of 1V produces an output of FFF.

In bipolar mode, the input voltage range is -0.5V to +0.5V. The output data word is in two's complement format, with an output of 800 for an input of -0.5V, an output of 000 for an input of 0V, and an output of 7FF for an input of +0.5V. The relationship between analog input voltage and digital ADC readings described here only applies to this device and is likely to be different in other FPGAs and microcontrollers.

The Arty board contains additional circuit elements to filter the analog inputs and to scale the voltage range received on some of those inputs. *Figure 2.3* shows the Arty circuitry associated with the analog input pins labeled A0-A5:

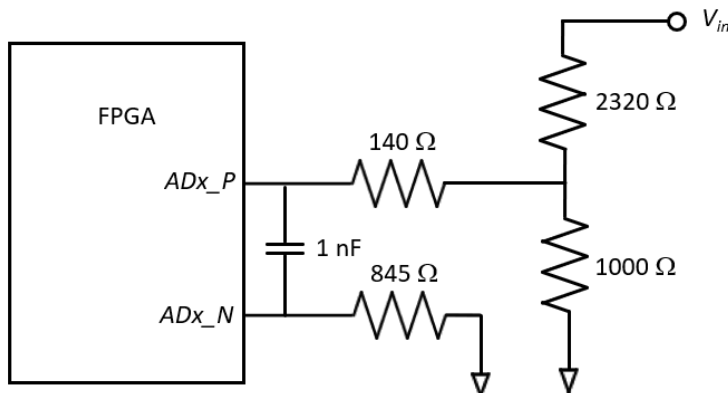


Figure 2.3 – Arty A7 ground-referenced analog input circuitry

In this diagram, the analog input signal, V_{in} , has a range of 0 to 3.3V. The resistor pair of 2,320 Ω and 1,000 Ω scale the voltage to the range 0-1V. The remaining resistors (140 Ω and 845 Ω) and the capacitor (1 nF) perform noise filtering on the input signal. Because the voltage V_{in} is referenced to ground, the negative signal of the differential pair (ADx_N) is connected to ground (indicated by the downward pointing triangles) through an 845 Ω filtering resistor. The positive and negative signals of the XADC differential pair are labeled ADx_P and ADx_N , where x indicates the ADC analog multiplexer input number.

The Arty boards also support differential analog inputs. The pins labeled $A5$ - $A11$ form three differential pairs, each with two noise filtering resistors and one capacitor, as shown in Figure 2.4:

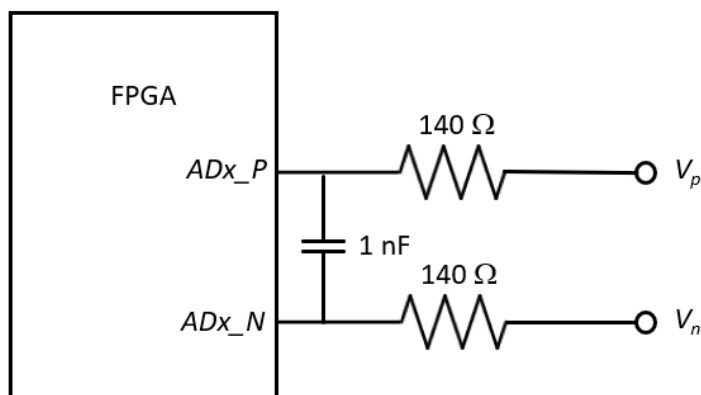


Figure 2.4 – Arty A7 differential analog input circuitry

In *Figure 2.4*, the analog input signal is defined as the difference signal V_p minus V_n , with a range of -0.5V to +0.5V.

This section provided a brief introduction to the concepts behind analog-to-digital conversion and the design of the analog input interfaces in the Arty A7 boards. The next section introduces a variety of sensor types used in embedded systems, some of which make use of interface circuitry similar to the circuits described in this section, and some of which require more sophisticated methods to interface with the host system.

The types of sensors used in embedded systems

This section provides a brief overview of a variety of sensor types used in embedded systems. This list is not exhaustive, but it should give you some idea of the variety of sensors available. It is the responsibility of the embedded system architect to identify the list of particular sensor types and the specifications those sensors must meet to implement any particular system design.

Light

Light sensors in embedded systems range in complexity from simple photoresistors to sophisticated multi-band sensor arrays used in devices such as video cameras, microscopes, and astronomical telescopes.

A **photoresistor** is a resistive device that exhibits a decrease in resistance as the **luminosity** (the intensity of light) increases on its surface. Photoresistors are commonly used in applications such as nightlights and for safety-related obstacle detection by automatic garage door openers.

Photodiodes and **phototransistors** are semiconductor devices that convert light into electrical current. These sensors are more sensitive than photoresistors and provide more consistent performance across a range of temperatures in comparison to photoresistors.

Video sensors contain a two-dimensional array of light-sensitive elements, often with filtering that limits the input to each element to a specific frequency range within the light spectrum. By providing separate sensors tuned to red, green, and blue colors, video cameras are able to capture the full range of colors visible to the human eye.

Temperature

As we saw in the *Introducing passive, active, and smart sensors* section earlier, a **thermistor** is a resistive element that changes resistance with changes in temperature. By using a voltage divider circuit, similar to the divider in *Figure 2.3* with a thermistor in place of the 2320 Ω resistor and a constant V_{in} of 3.3V, sensor resistance can be measured. The data sheet for a particular thermistor will provide the information needed to convert a resistance reading to the corresponding temperature.

A **thermocouple** is a temperature sensing device that connects two dissimilar metals at a single point. This results in a measurable voltage that can be compared to a reference voltage to determine the temperature at the sensor location.

Thermistors are most applicable for temperature ranges from -50 to 250 °C, while thermocouples support a more extreme temperature range of about -200 to 1,250 °C. Thermistors tend to be less expensive, while thermocouples are more complex devices requiring measurement of a tiny output voltage and the provision of a reference junction, which is a connection of the dissimilar metals at a point with a known temperature.

In most applications where an approximate temperature measurement is adequate, and the range of expected temperatures is limited, embedded system designs will typically use thermistors. For applications involving extreme temperatures, such as in ovens and furnaces, thermocouples tend to be the preferred sensor.

Pressure

Pressure sensors measure the pressure of liquids and gases. The pressure measurement may represent absolute pressure, meaning pressure relative to a perfect vacuum, or it may be relative to some reference such as the surrounding atmospheric pressure.

For example, a barometer, used to monitor weather conditions, measures absolute air pressure. An automotive tire pressure gauge, on the other hand, measures air pressure in the tire relative to the surrounding atmospheric pressure.

A *differential pressure sensor* measures the difference between pressure at two locations. Differential pressure sensors are used in applications such as measuring the pressure drop across an inline fluid filter.

Low-cost pressure sensors are commonly constructed of a *piezoresistive* material, which changes in resistance as stress is applied to a sensing element in response to the measured pressure. The embedded system measures the sensor resistance to determine the pressure reading.

Humidity

Humidity sensors measure the partial pressure of water vapor in the atmosphere. The reading is generally expressed as a percentage, which indicates the measured water vapor partial pressure relative to the saturation partial pressure (the maximum possible water vapor partial pressure) at the current temperature.

Humidity sensors are useful in environmental monitoring and control applications where it is desired to maintain humidity within desired limits to meet the needs of electronic equipment or for human comfort. A humidity sensor is often combined with a temperature sensor in a single unit.

Low-cost humidity sensors are constructed with a sensing element containing a polymer that varies in capacitance with changes in humidity. The sensor measures the capacitance of the sensor element, compensates the reading for the current temperature, and computes the relative humidity based on stored calibration information.

Some humidity sensors are smart sensors, containing on-board processing and digital communication capability. When using these devices, the host processor requests a reading from the sensor and retrieves the result once the measurement is complete.

Fluid flow

Fluid flow sensors measure the quantity of gas or liquid passing through the sensor's active region. These sensors usually operate by placing some type of restriction such as a screen or nozzle in the flowing fluid and measuring the effect of the restriction. By sensing the pressure drop across the restriction, it is possible to estimate the flow rate of the fluid.

Other measurement technologies that do not require flow restriction, such as ultrasonic and laser, are used in some sensor designs. Flow sensors are available with output in the form of an analog signal or via a digital interface.

Fluid flow sensors are used in a wide variety of applications in which it is important to accurately track the amount of fluid passing through a system. Embedded systems in automotive and aircraft applications use flow sensors to monitor the motion of fluids, including fuel, lubricant, and brake fluid. Medical applications use flow sensors for tracking the flow of fluids, such as anesthetics and in respiratory machines. The water meter and natural gas meter are two types of flow meters in common use in homes and commercial buildings.

Force

A **force sensor** measures the amount of force applied to an object. A bathroom scale is a good example of a force sensor. The scale measures the weight, which is a downward force, of the person standing on it.

Force sensors commonly use force-sensing resistors, also called strain gauges, which are constructed from material that changes in resistance as force is applied to it. Some types of force sensors operate on the piezoelectric principle or use the hydraulic or pneumatic displacement of a fluid or gas under pressure in response to the applied force.

Ultrasonic

As we saw earlier in this chapter, an ultrasonic sensor generates a sound wave at a frequency higher than the range of human hearing, transmits this signal into a measurement region, and listens for any echoes from objects that may be within range. The time between signal transmission and reception, multiplied by the speed of sound in the measurement medium (which may be air, some other gas, or a liquid), represents the round-trip distance from the transmitter to the target object and back to the receiver.

Many ultrasonic sensors use the same ultrasonic element to generate the transmitted pulse and to receive the echo signal. This helps to miniaturize the sensor assembly.

Simple ultrasonic sensors use two digital signals to control sensor operation and read the measurement output. The *Trigger* pin is a sensor input signal that initiates a measurement cycle in response to a rising pulse edge. The *Echo* pin is a sensor output that goes high when the pulse is transmitted and returns low when the echo is received. By measuring the time between the rising and falling edges of the *Echo* signal, the processor can determine the signal round-trip time, and from that it can compute the distance to the obstacle.

Audio

Audio sensors receive sound inputs in the range of human hearing and produce an electrical output in response to the received signal. A standard microphone is one example of an audio sensor. Intelligent assistant devices listen to sound in their environment continuously and, when a triggering sequence of syllables is detected (such as "Alexa" or "Hey Google"), the device records the following sounds and attempts to interpret the command provided as a sequence of words.

In simpler applications, an audio sensor may simply monitor the intensity of sound in its surroundings and produce an output when the sound level rises above a threshold. More sophisticated applications of audio monitoring include the sensing of glass breakage by a security system or the detection and localization of gunshots in urban areas.

Magnetic

A **magnetic field sensor**, or **magnetometer**, detects the presence of a magnetic field in the sensor vicinity. Simple magnetic field sensors respond only to the strength of the field. More sophisticated sensors measure the three-dimensional vector components of the magnetic field.

The Earth's magnetic field is produced by electric currents resulting from the flow of molten metal far below the surface. The strength and orientation of this field varies greatly at locations around the world, and the entire pattern of the field changes slowly from year to year.

A magnetic compass senses the Earth's magnetic field. A simple application of a magnetic sensor in an embedded system determines the sensor's orientation with respect to magnetic North. In a more sophisticated application, the system can calibrate the measurement based on its location on Earth, assuming it is known. Using a map of the Earth's magnetic field, the system can produce a more refined orientation estimate. This form of orientation sensing is susceptible to errors caused by the presence of ferromagnetic materials or other sources of magnetic fields.

In a home and office security system, sensors that detect the opening of a door often use a small magnet to generate a local magnetic field sensed by a switching element. A magnetic reed switch contains a flexible metal contact that closes when in close proximity to a small magnet. Opening the door separates the magnet from the switch, which opens the switch and notifies the alarm system that the door has opened.

Chemical

Chemical sensors measure attributes of chemical components in the vicinity of the sensor. These sensors are constructed to be sensitive to the presence of a particular element or compound within the gas or liquid surrounding the sensor.

Some common examples of chemical sensors include carbon monoxide detectors and radon detectors in residential settings. Modern gasoline engines contain oxygen sensors in the exhaust system that provide information that enables the fuel delivery system to deliver the optimal fuel-air mixture to the engine.

Low-cost, single-chip sensors are available to perform measurements of air quality, reporting the levels of carbon dioxide and volatile organic compounds in the surrounding air. **Nanotechnology** is providing a path to the production of a variety of chemical sensing devices across a broad range of applications. Because nanotube-based sensors are so tiny, it takes very few molecules of the gas being sensed to produce a measurable reading, resulting in sensors that are exceptionally sensitive and selective.

Ionizing radiation

Ionizing radiation consists of electromagnetic particles with sufficient energy to cause molecules and atoms to lose electrons when struck by the particles. Once a molecule or atom loses an electron, it becomes charged and thus becomes an ion. Ionizing radiation is commonly referred to as X-rays and gamma rays. Exposure to this form of radiation at low levels can be harmful to living tissue, resulting in long-term harm such as cancer. If received in a large enough concentrated dose, rapid tissue damage may occur.

Low-level ionizing radiation exists in the natural environment, either arriving from space (as cosmic rays and solar radiation) or from the Earth via the radioactive decay of elements such as uranium and thorium. Man-made ionizing radiation is produced by sources such as nuclear reactors and X-ray machines.

Sensors for ionizing radiation measure changes in a sensitive material resulting from incident radiation. Each measurable event consists of a particle interacting with the sensitive material. The resulting signal may be an electrical pulse, a burst of light, or in some sensors, a detectable change in a gas. Some ionizing radiation sensors report individual particle-triggered events while others measure and accumulate radiation dosage over time.

Radar

The term **RADAR** originated as an acronym for **Radio Detection and Ranging**. A radar system transmits radio frequency signals into its surroundings and receives echoes from any objects encountered by the pulses. By processing the received return signals, the radar system can determine the position of objects in its vicinity and, in some cases, derive other attributes such as the direction and speed of the object's motion.

Today, single-chip radar sensors are available for use in applications such as the sensing of nearby automobiles. Automotive radar sensors enable adaptive cruise control, which measures the distance and relative speed of a vehicle ahead of the sensor host vehicle. Using information provided by the radar sensor, the host vehicle maintains appropriate distance from the vehicle ahead of it and responds to events such as sudden braking by the lead vehicle.

Lidar

LIDAR is an acronym for **Light Detection and Ranging**. Lidar is similar in concept to radar, but instead of using radio frequency signals, lidar uses laser or infrared light to sense objects in the vicinity of the sensor.

A **lidar sensor** transmits a tightly focused pulse of light and measures the distance to an obstacle in the direction of the beam by sensing the time to reception of the reflection from the object. The sensor repeats this process perhaps thousands of times in rapid sequence, varying the pointing direction of the beam for each measurement, to build up a three-dimensional map, referred to as a point cloud, of the terrain and objects in the sensor field of view.

Precision lidar sensors require costly optical components for the laser transmitter and sensor, which has traditionally meant these sensors were quite expensive. In recent years, however, with the growth of autonomous vehicle technology, the price of lidar sensors has decreased substantially.

Video and infrared

A traditional digital color video camera produces a sequence of images, each composed of a two-dimensional array of pixels. Each pixel contains three color intensities (red, green, and blue), which permits representation of colors across the visible spectrum.

An embedded system can use a video camera to capture a scene for human consumption, such as in a video doorbell application. For this usage, the camera must produce an image with sufficient resolution (usually at least a few hundred pixels in both the horizontal and vertical dimensions) and at an update rate that creates the perception of smooth motion in the scene, typically 30 frames per second, though slower update rates are acceptable in some applications.

Some application areas require machine processing of video data streams. One relatively simple application of video data processing is an area monitoring security system. When this system is armed, it expects to see no activity at all in the monitored area. If motion is present in the video scene, the system can detect it by comparing sequential video frames and looking for differences in the images.

Autonomous vehicles require a more sophisticated video processing system than simple frame differencing. These vehicles must integrate several types of sensor data, which often includes video cameras, to determine the presence of roadways, traffic signs, vehicles, pedestrians, bicycles, and any other type of object or obstacle that a human driver would be expected to deal with safely. While the video camera sensor portion of an autonomous vehicle is similar in concept to the camera in a simple video doorbell, the processing of the video data in an autonomous vehicle is far more complex, requiring high-performance, real-time processing using sophisticated artificial intelligence algorithms.

Infrared cameras are similar to video cameras, producing a sequence of two-dimensional images. The primary difference is that infrared cameras are sensitive to light wavelengths that are longer than the wavelength of the color red, which is the longest wavelength visible to the human eye. Infrared sensors can respond to the temperature of an object because heat energy is radiated in the infrared band. Imaging infrared sensors are suitable for applications such as monitoring a circuit board for hot spots while in operation.

Inertial

An **inertial sensor** detects changes to the motion of the body to which it is attached. Specifically, a single inertial sensor measures acceleration along an axis or the rotational rate about an axis. To fully characterize the acceleration of an object in three-dimensional space requires three accelerometers with measurement axes aligned along orthogonal X, Y, and Z axes. Similarly, full characterization of rotational motion requires three rotational rate sensors about the same axes.

Inertial sensors are used in aircraft, spacecraft, and ships to accurately track motion in the presence of disturbances such as turbulence and maneuvering. To operate properly, an inertial navigation system must be initialized with its correct starting position, and measurement errors associated with the inertial sensors must be small enough that they do not degrade the position measurement to an unacceptable degree.

GPS

Global Positioning System (GPS) receivers collect signals from a constellation of satellites in orbit around the Earth and use the information in the signals to compute the position of the receiver as well as provide the current time. Modern low-cost, single-chip GPS receivers can determine their location anywhere on Earth to an accuracy of a few meters and report the current time with microsecond accuracy.

In addition to the American GPS system, several other satellite navigation constellations are operational: Galileo (European Union), Beidou (China), and Glonass (Russia). While the signals used by each of these systems are not directly compatible, modern satellite navigation receivers, including low-cost, single-chip designs, are capable of receiving signals from some or all of these constellations simultaneously. The advantages of a multi-constellation receiver are faster time to first fix (the first accurate position measurement after the receiver is powered on) and better position measurement accuracy because of the higher likelihood of receiving signals from satellites at suitable geometrical locations.

Today, **Global Navigation Satellite System (GNSS)** receivers are commodity items found in aircraft, automobiles, farm vehicles, surveying equipment, military systems, smartphones, and even pet collars. Any embedded system that operates outdoors, or that has even intermittent access to a view of the outdoors through a window, can potentially incorporate a GNSS receiver to accurately determine its position and the current time.

The most sophisticated application of GNSS receivers in navigation applications integrates a suite of inertial sensors to measure three-axis accelerations and rotational rates. A GNSS receiver has a relatively large error in each individual measurement but the system can provide a much more precise position by averaging a large number of sequential measurements. An inertial sensor, in comparison, provides precise information about changes in position from update to update, but is subject to long-term drift and the corresponding accumulation of errors.

By carefully integrating a GNSS receiver with an inertial sensor suite, the combined system takes advantage of the best features of each sensing subsystem and cancels out the largest errors produced by the other subsystem. In effect, this sensor assembly, referred to as a **GPS/INS system**, uses the inertial sensors to compute high-rate updates to the system's position and angular orientation while simultaneously using the GNSS measurements to correct for the biases and other error sources in the inertial sensor measurements. GPS/INS systems are used widely in aircraft navigation, ship navigation, and in military applications.

The next section will present some of the most useful communication technologies for connecting sensors to embedded processors.

Communicating with sensors

In the previous section, we looked at a variety of sensor types suitable for measuring various attributes of an embedded system and its environment. As part of each sensor measurement, the sensed data must be forwarded to the system processor. This section examines the most common interface technologies used in embedded systems for communication between sensors and processors.

GPIO

A **General-Purpose I/O (GPIO)** input signal is simply a physical pin on the processor that, when read, indicates whether the voltage at the pin is low (near 0V) or high (near the upper end of the processor I/O voltage range, often 5V or 3.3V). GPIO inputs can be used to detect operator actions such as button presses, or to determine whether the system is in an unsafe condition, perhaps by using a switch to detect when a safety-critical cover has been opened.

A GPIO input signal can be used with an analog sensor to detect when the analog signal is above or below a threshold value. The circuit in *Figure 2.5* uses a comparator to detect whether the light level measured by a photoresistor is above a threshold:

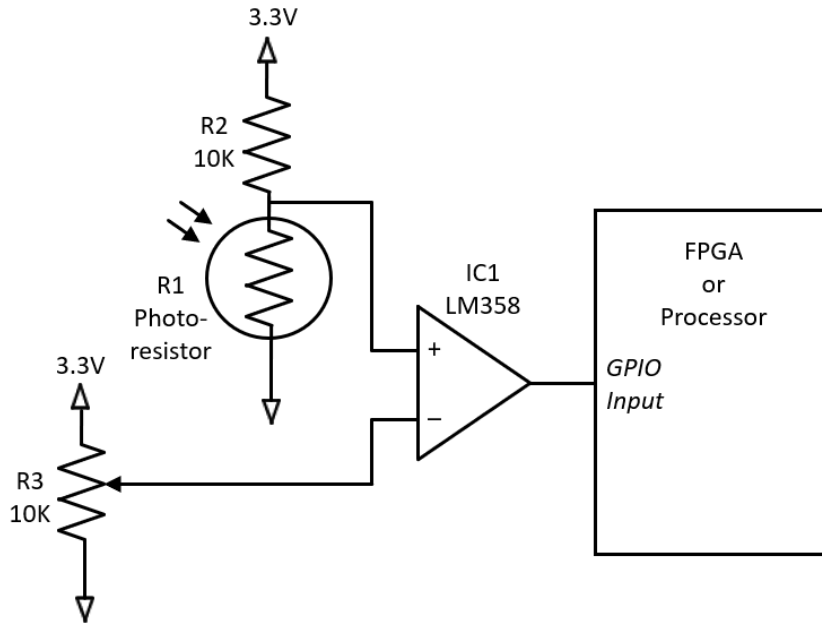


Figure 2.5 – Light detection circuit

A **comparator** is an electronic device that, in effect, subtracts the voltage at its – input from the voltage at its + input and outputs a high level (3.3V in this case) if the sign of the difference is positive, or a low level (0V) if the difference is negative. In this application, a comparator is a one-bit analog-to-digital converter. The LM358 is a standard 8-pin integrated circuit containing two operational amplifiers that are suitable for use as comparators.

In the circuit of *Figure 2.5*, **R1** is a phototransistor that varies in resistance as the level of illumination (indicated by the two arrows pointing toward **R1**) varies. This results in changes in the voltage at the connection point between **R1** and **R2**, which feeds into the comparator + input.

R3 is a **potentiometer**, an adjustable resistor. In the configuration of *Figure 2.5*, adjusting **R3** sets the voltage presented to the IC3 – input to any fixed voltage between 0V and 3.3V. This voltage sets the threshold at which the output of **IC1** changes state.

The circuit of *Figure 2.5* has a limitation that may result in undesirable behavior. If the voltage on the **IC1** + pin slowly approaches and passes through the voltage on the – pin, the ubiquitous presence of noise on the two analog inputs to **IC1** may result in undesired flickering as the **IC1** output toggles between high and low while the two voltages are very close together. As with many situations in embedded system design, there are two ways to approach this problem: Fix it in hardware or in fix it in software:

- **Hardware solution:** By adding two resistors, as shown in *Figure 2.6*, hysteresis can be added to the comparator switching logic. **Hysteresis** introduces a dependence of the switching logic on its past switching behavior. The two resistors in this diagram add a small offset voltage to the **IC1** + input. The sign of this offset changes when the output of **IC1** changes state. The result is that after a slowly varying input causes the output to switch from high to low, the input voltage has to backtrack some distance in the other direction before the output will switch back to the high level. As long as the distance it must backtrack in order to change the output is less than the analog noise on the inputs, there will be no flickering of the output signal:

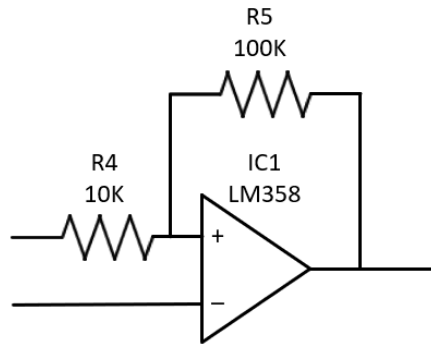


Figure 2.6 – Comparator with hysteresis

- **Software solution:** The drawbacks of the hardware solution of *Figure 2.6* include additional circuit complexity, more parts required, and the need for a calibration operation to select the best values of **R4** and **R5** for the application (the resistor values in *Figure 2.6* are just examples). Because the output of **IC1** in *Figure 2.6* feeds into a digital processor, a software filtering algorithm can be applied to eliminate the flickering instead of relying on a hardware solution.

One fairly simple approach is to count the number of sequential readings of the **IC1** output that are different from the most recent known valid reading of that signal. If we require that a number (perhaps 100) of sequential readings of the input must be identical before changing the "valid" state of the measurement, flicker should be largely eliminated.

The C language listing below presents code suitable for use in the **Arduino** (<https://www.arduino.cc/>) environment that initializes the reading from a GPIO input pin by averaging several readings, and then uses a count of consecutive readings to determine when the state of the input has changed. The Arduino executive calls the `setup()` function one time at system reset and then repetitively calls the `loop()` function during system operation. The value of the `switchCount` constant must be carefully chosen to be large enough to avoid responding to occasional sequential noise-induced errors (resulting in flickering), but small enough not to make the response to a varying input too sluggish:

```
const int lightInputPin = 7; // Light sensor is on pin 7
const int switchCount = 100; // Count at which to switch

int lightState = LOW; // The current validated light state
int lightCount = 0; // Counter of sequential readings

void setup() {
    pinMode(lightInputPin, INPUT); // Set pin 7 to be an input

    // Read lightInputPin repeatedly to get an average
    lightCount = 0;
    for (int i=0; i<switchCount; i++)
        if (digitalRead(lightInputPin) == HIGH)
            lightCount++;
        else
            lightCount--;

    // If more reads were HIGH than were LOW, set state to HIGH
    lightState = (lightCount > 0) ? HIGH : LOW;
    lightCount = 0;
}

void loop() {
    // If this reading matches lightState, reset lightCount
    // Count up if this reading differs from lightState
```

```
if (digitalRead(lightInputPin) == lightState)
    lightCount = 0;
else
    lightCount++;

// Toggle lightState when switchCount is reached
if (lightCount >= switchCount)
{
    lightState = (lightState == HIGH) ? LOW : HIGH;
    lightCount = 0;
}
```

We will not be working with Arduino systems in this book, but this code is a workable example you can run on an Arduino system to demonstrate the filtering provided by the algorithm described here.

Most embedded system architectures make use of at least a few GPIO input signals, either as detectors of events such as switch closings, or as status inputs from other digital components in the system.

Analog voltage

Analog signals are sensed using ADCs in the manner discussed in the *Applying analog-to-digital converters* section in this chapter. Integrated ADCs in microcontrollers and FPGAs provide a moderate level of performance, both in terms of maximum sample rate and the number of bits per sample. The interface between processor instructions and the ADC in these devices is, in general, a collection of registers used to configure the ADC, initiate a measurement, and receive the converted digital reading after the measurement completes. Integrated ADCs often provide additional features, such as the ability to trigger a processor interrupt upon completion of a conversion, and to automatically initiate conversions at a fixed rate.

While integrated ADCs are adequate for sampling signals at modest rates, some applications require ADC measurements at much higher rates, in the hundreds of millions of samples per second up to billions of samples per second. Integrated ADCs might provide 10 or 12 bits of resolution, but application requirements may demand 14 or 16 bits of precision.

In applications that require these high sample rates and high precisions, a dedicated ADC with the required performance specifications must be used. Because of the extremely high sample rate of these ADCs, possibly combined with a large number of bits per sample, the data output rate from the ADC can be as high as several billion bits per second. Two of the interface architectures designed to address the data rate requirements of high sample rate ADCs are serial LVDS and JESD204:

- Serial **Low Voltage Digital Signaling (LVDS)** is a 2001 standard for interfacing high sample rate ADCs to FPGAs and to **Digital Signal Processors (DSPs)**. Serial LVDS uses differential signal pairs to perform high-speed data transmission. A serial LVDS interface may contain a single differential signal pair, referred to as a **lane**, or it may contain multiple lanes that transfer data bits simultaneously. Each serial LVDS transmitter outputs 3.5 mA of current that produces 350 mV across a 100 Ω resistor at the receiver. The transmitter outputs current continuously and switches the direction of current flow to generate clock signals and data bit signals. Serial LVDS can support single-lane data rates up to about 1 **billion bits per second (Gb/s)**. An example of a device supporting serial LVDS is the Analog Devices HMCAD1511 8-bit ADC (<https://www.analog.com/media/en/technical-documentation/data-sheets/hmcad1511.pdf>), which is capable of sampling four separate signals at 250 MSPS or one signal at 1 **billion samples per second (GSPS)**.
- The JEDEC *JESD204* standard, most recently updated to JESD204C in 2017, specifies a serial interface between ADCs and digital processors. Like serial LVDS, JESD204 supports single- and multiple-lane differential signal pairs. The most striking difference from serial LVDS is that a JESD204C lane supports data rates up to 32 Gb/s. Devices that comply with JESD204C support several additional features that are not available with serial LVDS, including synchronization across multiple lanes and synchronization across multiple ADCs.

In applications requiring high sample rate ADC input, it is common to connect the ADC to an FPGA using a serial LVDS or JESD204 interface. The FPGA, when programmed with a suitable algorithm, is capable of receiving the high-rate incoming data and performing the initial stages of processing using the parallel hardware resources of the FPGA. The processed data stream output by the FPGA is generally reduced in size substantially by compression, filtering, or other algorithms within the FPGA. The FPGA forwards this smaller data stream to the system processor.

I2C

For sensor interfaces that do not require high data rates, simplicity and low production cost become primary concerns. For low data rate communication between devices on a single circuit board, or for communication among multiple circuit boards within an enclosure, the **Inter-Integrated Circuit (I2C)** bus architecture is a popular choice.

The I2C interface consists of two open collector lines pulled up by resistors: a clock line and a data line. An **open collector** is a type of digital output signal that has the useful property that, when activated, pulls the signal line low, but when it is inactive, it does not drive the signal line at all. This allows the pullup resistor to bring the signal level high. The term *open collector* applies to NPN transistors, while a similarly functioning MOSFET transistor is referred to as an **open drain**.

The use of open collector (or open drain) devices on an I2C bus allows several devices to be connected on the same pair of signal lines. Each device monitors the state of the signal lines and only activates its output when it needs to communicate on the bus.

I2C implements a **master-servant** network architecture. An I2C network can contain several nodes, but only one node can be the master at any time. All activity on the bus is managed by the master, which uses a unique 7-bit address assigned to each servant node to communicate with that device.

The master generates the clock signal and initiates command sequences as serial data transfers on the data line. A command includes a servant address, possibly a register address within the servant device, and an instruction for the operation to be performed. Data can transfer in either direction between the master and the addressed servant. Because there is only one data line, data can move in only one direction at a time, making I2C a **half-duplex** communication protocol.

After a command and any associated data transfer have completed, a different node can become the master and begin issuing a clock signal and commands on the bus. Multi-master operation must be coordinated to ensure there is only one active master at any time. *Figure 2.7* shows a simple I2C bus with one master and two servants. The common names for the I2C signals are **SCL** (serial clock) and **SDA** (serial data):

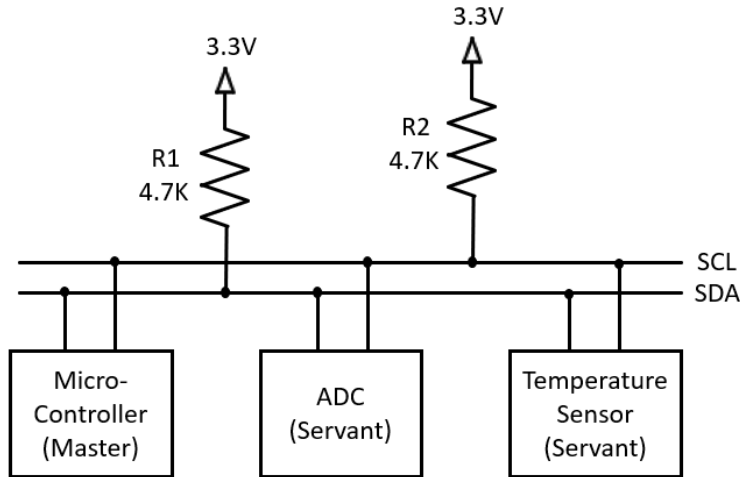


Figure 2.7 – Simple I2C bus architecture

Common data clock speeds for I2C buses are 100 Kb/s and 400 Kb/s, though recent revisions of the I2C standard can operate at up to 5 Mb/s.

I2C interfaces are widely used with sensors that can work within the data transfer speed limitation of the bus. Some examples of sensors with I2C interfaces are ADCs, pressure sensors, temperature sensors, GPS receivers, and ultrasonic sensors.

SPI

The **Serial Peripheral Interface (SPI)** is a four-wire serial data bus. A single node, called the **master**, manages activity on the bus. Other nodes, which we will call **servants**, respond to commands from the master. The four wires of the bus perform these functions:

- The **chip select (CS)** signal notifies a servant that the master is interacting with it.
- The **serial clock (SCLK)** signal sequences the transfer of data on the bus, one bit per clock cycle.
- The **master-in-servant-out (MISO)** line transfers data from the master to the servant.
- The **master-out-servant-in (MOSI)** line transfers data from the master to the servant.

SPI can transfer data in both directions simultaneously on the MISO and MOSI lines, making SPI a **full-duplex** communication standard.

Figure 2.8 shows an SPI bus with the same node types as Figure 2.7:

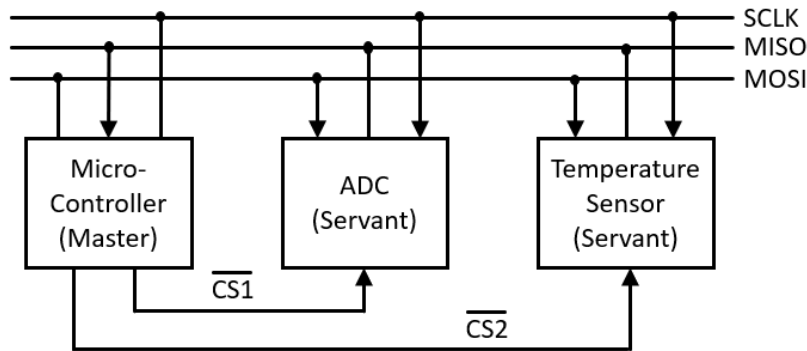


Figure 2.8 – Simple SPI bus architecture

In addition to requiring a minimum of twice the number of signals as the I2C bus, an additional chip select signal is required for each additional servant node on the bus. Despite the additional hardware support, the SPI architecture offers a number of advantages over I2C:

- Higher clock speeds are possible. SPI buses operating at clock speeds up to 50 MHz are common.
- The full-duplex nature of SPI permits a potential doubling of the data transfer rate at a given clock speed compared to I2C.
- There is no need to manage servant addresses.
- SPI tends to use less power because there are no pullup resistors dissipating energy.

In general, if you have a choice between I2C and SPI, you may prefer to use SPI when data transfer speed is critical and power consumption is a concern. On the other hand, I2C can communicate with a larger number of peripherals, though usually at a lower data rate.

CAN bus

The **Controller Area Network (CAN)** bus is a serial data bus designed to operate in the harsh environment of automobiles. One of the primary purposes of the CAN bus is to reduce the amount of wiring in motor vehicles. For example, an automotive taillight assembly might contain lights with several different functions (brake light, backup light, driving light, turn signal). In an analog implementation, a separate wire would be required to operate each of these lights. In a CAN implementation, all that is required is wiring for power to the module and the CAN bus connection. A microcontroller in the taillight module activates each of the lights in response to digital messages received over the bus.

In comparison to the I2C and SPI bus architectures, CAN is a much more sophisticated communication architecture supporting prioritized message delivery and multiple error detection and recovery mechanisms.

A CAN bus consists of a potentially large number of nodes connected along a differential wire pair. Because the standard CAN bus can operate at fairly high bit rates (up to 1 Mb/s) over long bus lines (up to about 25 meters at 1 Mb/s), the differential bus pair must be terminated at each end with a $120\ \Omega$ resistor. The resistors match the impedance of the cable and prevent reflected signals from traveling back down the line.

Signaling on the two wires of the differential pair in a CAN bus is similar in concept to the open collector drivers on an I2C bus. When none of the nodes on the bus are transmitting, both bus lines are pulled to a nominal level of 2.5V. This is referred to as the **recessive state** and represents a logical data value of 0. When any node on the bus transmits a logical 1, called the **dominant state**, it pulls one of the bus lines (named CAN_H) to a higher voltage, around 3.5V, and the other (CAN_L) to a lower voltage, around 1.5V. Figure 2.9 shows a simple CAN bus architecture:

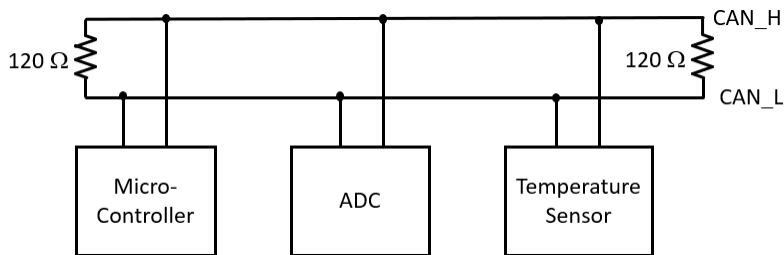


Figure 2.9 – Simple CAN bus architecture

The CAN bus does not have a master node like the I2C and SPI buses. All the nodes in a CAN bus are peers and any of them can initiate the transmission of a message at any time the bus is idle. If multiple nodes attempt to send messages at the same time, perhaps after waiting for the transmission of a previous message to complete, an automatic prioritization scheme allows the highest priority message to proceed and any other nodes attempting to transmit must stop and wait for the next idle period.

Each CAN message begins with an identifier that is either 11 or 29 bits in length. Recipients of messages examine the identifiers in messages they receive to determine which messages to process. The message identifier field determines the message priority, with numerically lower identifiers having higher priority. The prioritization process supports real-time interactions among subsystems by ensuring the highest priority messages get the first chance to access the bus.

Wireless

Increasing numbers of embedded systems are using wireless technologies to communicate with supervisory systems and across the internet. The primary benefit of wireless communication for end users is the elimination of the need for data cabling. As long as power is available for an embedded system and it is within range of a wireless communication node, no further installation work is needed to enable communication.

The communication range of a wireless technology can range from very short-range (a few centimeters) to spanning the globe. The costs, power requirements, and data transfer rates vary greatly among the available solutions. Some of the most common wireless communication technologies used by embedded systems, ordered from the shortest to the longest communication range, are as follows:

- **Radio frequency ID (RFID):** This technology uses tags attached to objects or in badges carried by humans to provide a unique identifying code to a reader. Passive RFID tags derive power from an electrical field produced by the reader and transmit a signal that is received by the reader. Active tags use a power source, such as a battery, to power the tag. Passive tags must normally be very close to a reader, if not touching it, to produce a successful reading. Active tags may operate at a significant distance, perhaps in the hundreds of meters, from a reader.
- **Bluetooth:** Bluetooth is widely used in smartphones, automobiles, and in other smart devices. Embedded systems can take advantage of Bluetooth connectivity in usage scenarios that are compatible with the features and limitations of Bluetooth. Bluetooth operates over a short range, typically 30 meters or less. **Bluetooth Low-Energy (BLE)** is optimized for applications that send data in short bursts at up to hundreds of kilobits per second while consuming very little power. If an embedded system application is compatible with the capabilities of Bluetooth or BLE, it is straightforward to integrate a single-chip communication solution into the design.
- **Wi-Fi:** Wi-Fi is the name of a family of wireless networking protocols used in local networks. The range of Wi-Fi communications depends on factors such as the presence of walls or other obstacles between two Wi-Fi nodes. In a home environment, the communication range might be limited to a single room or between adjacent rooms. Outdoors, communication may work well at distances of 100 meters or more. Wi-Fi is intended to support high-speed networking, and, with a strong signal connection, is capable of transferring hundreds of megabits per second. Wi-Fi is commonly used in embedded applications such as video doorbells.

- **Cellular:** Cellular network communication is suitable for embedded system architectures that require access to wide area network communication from anywhere within range of a cellular network carrier antenna. While cellular communication enables interaction with embedded devices around the world, the downside of cellular is the need for a SIM card and a data plan associated with each device. If your application has a need for widespread wireless connectivity, such as tracking fleet vehicle movements, cellular networking is the way to go.

This chapter has listed a variety of types of embedded system sensors and some of the communication technologies used to transfer sensor data to higher-level processing, either within the host system containing the sensor or to a supervisor node accessible via a network. The next section will briefly introduce some standard processing methods used to transform raw sensor data into actionable information for use in an algorithm.

Processing sensor data

Sensors measuring various attributes of a system, such as the pressure in a pipe or the air temperature, will generally contain some amount of error. Sensor measurement errors have a variety of causes, including sensor calibration inaccuracies, non-ideal measurement configurations, temperature dependence of the sensor or its associated circuitry, and the background noise that is always present in electronic circuits.

In some cases, particularly with non-critical measurements, it may not be necessary to take steps to compensate for measurement error. In many applications, however, it is critical to eliminate as much of the error as possible.

Precision sensors tend to be more costly than generic sensors, but the higher price typically brings with it a variety of techniques within the sensor design that improve its measurement quality. These techniques include features such as compensation for temperature variation and precision factory calibration.

Sometimes it is necessary for system developers to perform additional calibration to improve measurement accuracy. For example, a simple method to calibrate a temperature sensor is to immerse it in well-mixed ice water to get a reading for 0 °C and immerse it again in boiling water to get a reading at 100 °C. These two measurements can be used to construct a calibration curve that returns a more accurate reading than relying on the sensor's data sheet alone.

For sensor measurements containing additive random noise, simply averaging several readings will cancel much of the noise. This assumes the underlying signal does not vary significantly during the series of measurements. In situations where it is necessary to reduce random noise in rapidly varying signals, it may be useful to construct a digital filter to process the raw input signal. A simple, but non-optimal, digital filter simply averages the last N measurements at each update. Far more sophisticated filtering methods are available using digital filter design procedures, though those methods are beyond the scope of this discussion.

Summary

This chapter introduced a variety of sensors used across a wide range of embedded applications. We saw that passive sensors measure attributes of the world, such as temperature, pressure, humidity, light intensity, and atmospheric composition, while active sensors use technologies such as radar and lidar to detect objects and measure their position and velocity. This chapter also discussed the types of processing an embedded system must perform in order to convert raw sensor readings into actionable data.

Having completed this chapter, you have learned about the different types of sensors used in embedded systems and understand what passive and active sensors are, and are familiar with several types of sensors in each category. You are also familiar with the basic processing techniques performed on raw sensor data to provide information suitable for use in processing algorithms.

The next chapter discusses the need for embedded systems to generate real-time responses to inputs measured from sensors. The concepts of **real-time operating systems (RTOSes)** are introduced and the differences between an RTOS and a general-purpose operating system are discussed. The chapter will present the key features of some popular open source RTOS implementations and describe the ways in which these capabilities enable responsive and reliable embedded system architectures.

3

Operating in Real Time

This chapter addresses the need for embedded systems to generate real-time responses to inputs from sensors and other sources. The concepts of **Real-Time Operating Systems (RTOSes)** and their key features are introduced, as well as some challenges that commonly occur when implementing multitasking real-time applications. The chapter concludes with a discussion of the important characteristics of some popular open source and commercial RTOS implementations.

After completing this chapter, you will understand what it means for a system to operate in real time and will know the key attributes a real-time system must exhibit. You will understand the RTOS features that embedded systems rely upon, as well as some problems that frequently crop up in real-time embedded system designs. You will also have learned the key features of several popular RTOS implementations.

We will cover the following topics in this chapter:

- What does *real-time* mean?
- Attributes of a real-time embedded system
- Understanding key RTOS features and challenges
- Popular RTOSes

Technical requirements

The files for this chapter are available at <https://github.com/PacktPublishing/Architecting-High-Performance-Embedded-Systems>.

What does real-time mean?

Real-time means computing with a deadline. In a real-time embedded system, the time it takes to respond to an input is a critical component of system performance. If the system produces a correct response, but the response is not generated within the required time limit, the effect may range from a mild nuisance to a catastrophic impact for a safety-related system.

The response of a real-time embedded system to inputs must be both correct and timely. Most standard software development approaches focus on the correctness of the response produced by a piece of code rather than being overly concerned with the timeliness of the response. Non-real-time software development approaches attempt to develop code that executes as quickly as possible, but usually do not provide a hard time limit specifying when the response *must* be provided. Real-time systems are considered to have failed if the timing constraints are not met, even during stressing and rare combinations of operating conditions. A computing system that produces the intended outputs is considered *functionally correct*. A system that produces outputs within specified time limits is considered *temporally correct*. Real-time systems must be both functionally and temporally correct.

Consider two automotive embedded subsystems: the digital key fob used to unlock the car door and the airbag control system. If the key fob takes a few seconds longer for the car door to unlock than expected, the user may be a bit irritated, but will still be able to get into the car and operate it. But, if the airbag controller were to take a fraction of a second longer to respond than expected in a serious collision, the result may be passenger fatalities.

Real-time applications can be divided into two categories: soft real-time and hard real-time. Systems in which real-time behavior, defined as the ability to meet all of the system's timing requirements, is highly desired but not absolutely necessary are called **soft real-time systems**. The automotive key fob response time is an example of this category. While undesired delays in response may have negative impacts, such as reducing the level of perceived product quality in users' minds, the system nevertheless remains functional and usable. Real-time systems that must, under all circumstances, strictly meet all of their timing requirements, such as the airbag controller, are considered **hard real-time systems**.

The process used to develop and test software for embedded applications must maintain a continuous focus on the system's real-time requirements and ensure the software implementation does not compromise performance in terms of those requirements. For example, if noisy sensor measurements require digital filtering to reduce the effects of the noise, the code to implement the filtering will most likely require the insertion of loops in the code to implement the algorithm. The addition of loops, particularly if they iterate a large number of times, can substantially increase code execution time and possibly violate timing requirements.

The next section will examine the key attributes a real-time embedded system must possess, including the necessary features of the processor hardware, I/O devices, and operating system-level software.

Attributes of a real-time embedded system

The hardware and software of a real-time embedded system must exhibit some specific characteristics to ensure the system reliably meets its performance goals of producing reliably correct and timely outputs. Most real-time embedded systems that perform functions of moderate to high complexity must divide the processing work into multiple tasks that execute in an apparently (to the user) simultaneous manner, including managing the operation of hardware such as an automobile engine while regularly updating information displayed to the driver.

At the finest-grained level of processor operation, most embedded systems rely on the use of interrupts to notify the processor when an operation is required by an I/O device. In a real-time application, the handling of interrupts can become a critical factor in ensuring proper system operation. At the simplest level, any time an interrupt is being processed, the code algorithm that was paused to handle the interrupt is blocked from execution. This means that when the paused code resumes execution, it will have less time to complete before its deadline arrives. As a rule of thumb, it is best to minimize the amount of time spent handling interrupts.

Related to interrupt processing, the time-related performance of I/O devices is another important factor in the performance of real-time applications. Some I/O devices, such as a flash memory card, may require a substantial length of time to complete a read or write operation. When working with these devices, it is not acceptable for the processor to stop and simply wait for the operation to complete. Similarly, an ADC takes some time to perform the analog-to-digital conversion operation. If the processor spins on the *conversion complete* status bit, waiting for the conversion to finish, the delay may again be unacceptable. More sophisticated techniques are required when working with these devices.

The following sections discuss these system concerns and the important real-time performance attributes associated with each of them.

Performing multiple tasks

It is common for an embedded system to appear to be performing multiple tasks simultaneously. It is normally not necessary for the system to perform multiple different functions at the same precise point in time. Instead, it is generally acceptable to rapidly switch from performing one task, to the next, and so on. If each task succeeds at updating at its intended rate, it does not matter whether the system performs other actions between those updates.

It is also common for the various tasks a system performs to require updates at different rates. For example, a system that controls the speed of a vehicle electric drive motor may need to update the outputs that control the motor dozens of times per second, while the same device updates status information displayed to the user just a few times per second.

It is certainly possible for a developer to combine the code to perform both of these tasks (motor control and status display) with code that manages the execution of each task at appropriate time intervals, all within a single module. However, this is not an ideal approach. It is conceptually simpler to break the application code for each task into a logically separate module and manage the scheduling of the tasks from a higher-level module.

To provide a concrete example, assume we need to update an electric motor control task at a 50 Hz rate and update the user status display at a 10 Hz rate. We will also assume the longest possible time it takes the motor control code to run is 5 ms and the user status display code takes up to 10 ms to run, due to the need to transfer data over a slow interface. If we reach a point where both tasks are ready to run, we must ensure the motor control task receives the highest priority because we need the motor updates to execute at precise time intervals. Updating the status display is lower priority because if the timing of updates to the status display varies by a few milliseconds, it will not be noticeable to the user. In this example, the motor control task has a hard real-time requirement, while the status display task is a soft real-time function.

The following C language listing is an example of a control program that initializes the system, and then executes a loop at 20 ms intervals, updating the motor control on each pass. Every fifth pass through the loop, it also updates the status display after completing the motor control update. In this code, the `WaitFor20msTimer` function may be implemented as an interrupt-driven function that places the processor in a low power sleep state while waiting for the timer interrupt to wake it.

Alternatively, `WaitFor20msTimer` may contain a simple loop that reads a hardware timer register until the timer reaches the next 20 ms increment, at which point it returns:

```
void InitializeSystem(void);
void WaitFor20msTimer(void);
void UpdateMotorControl(void);
void UpdateStatusDisplay(void);

int main()
{
    InitializeSystem();

    int pass_count = 0;
    const int status_display_interval = 5;

    for (;;)
    {
        WaitFor20msTimer();

        UpdateMotorControl();

        ++pass_count;

        if (pass_count == 1)
        {
            UpdateStatusDisplay();
        }
        else if (pass_count == status_display_interval)
        {
            pass_count = 0;
        }
    }

    return 0;
}
```


This code executes in the pattern shown in *Figure 3.1*. The Motor Control code executes for 5 ms at 20 ms intervals, represented by the pulses in the upper graph. After the first Motor Control update completes, the control loop calls the Status Display update routine at time A. The dashed line in the diagram shows this relationship between the end of motor update processing and the start of Status Display update processing, which appears in the lower graph:

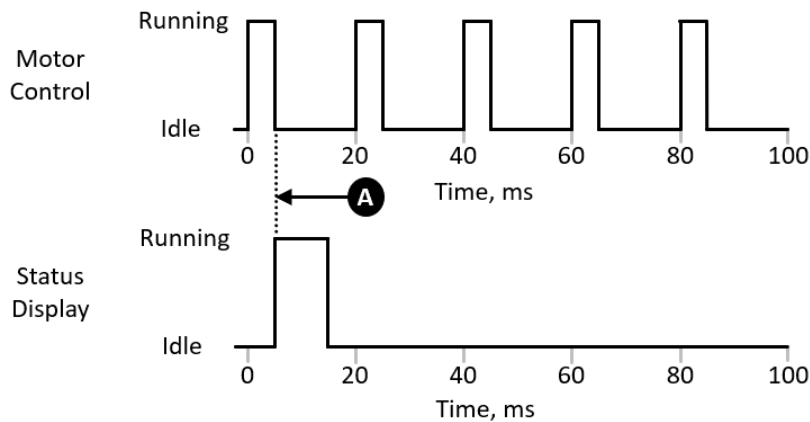


Figure 3.1 – Embedded system control loop timing

This code will be guaranteed to meet its timing requirements for Motor Control updates and Status Display updates as long as the processing time for each of the update routines remains within its constraints. There will be some small amount of timing jitter on the Status Display updates because the Status Display routine begins after the Motor Control update ends, and there is no guarantee that the time the Motor Control code takes to execute will be identical each time it runs.

What happens, though, if the Status Display code is upgraded to pass additional information to the display and the new version takes 20 ms to execute instead of 10 ms, as in the original version? From *Figure 3.1*, we can see that execution of the Status Display update will stretch 5 ms into the time period intended for the Motor Control update, delaying its execution. We have already determined that this sort of delay is unacceptable. What can we do to resolve this problem?

One possible approach is to split the Status Display update code into two separate routines, each taking no more than 10 ms to execute. These routines can be called in sequence, as shown in *Figure 3.2*:

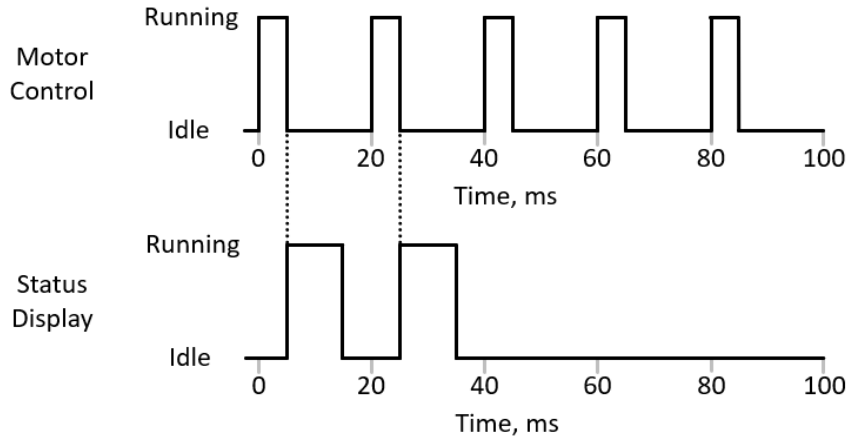


Figure 3.2 – Status update split into two parts

This solution will continue to meet all of the timing performance requirements, as long as each stage of the status update code finishes execution within its 10 ms time limit. The following code listing implements this solution:

```
void InitializeSystem(void);
void WaitFor20msTimer(void);
void UpdateMotorControl(void);
void UpdateStatusDisplay1(void);
void UpdateStatusDisplay2(void);

int main()
{
    InitializeSystem();

    int pass_count = 0;
    const int status_display_interval = 5;

    for (;;)
    {
        WaitFor20msTimer();

        UpdateMotorControl();
```

```
    ++pass_count;

    if (pass_count == 1)
    {
        UpdateStatusDisplay1();
    }
    else if (pass_count == 2)
    {
        UpdateStatusDisplay2();
    }
    else if (pass_count == status_display_interval)
    {
        pass_count = 0;
    }
}

return 0;
}
```

In this version of the application, the Status Display code is broken down into two functions: `UpdateStatusDisplay1()` and `UpdateStatusDisplay2()`.

While this solution is workable in terms of meeting the timing requirements for the system, it is far from an ideal approach. For one thing, it may not be easy, or even possible, to separate the Status Display update code into two functions, each taking approximately the same length of time to execute. Ongoing maintenance becomes more of a problem when changes must be made to this code. In addition to ensuring the new code is functionally correct, it must be distributed between the two update functions to ensure neither exceeds its execution time limit. This is, frankly, a brittle solution.

For a real-time embedded system design, this approach is clearly inappropriate. Much of the complexity in this example can be avoided through the use of preemptive multitasking. **Preemptive multitasking** is the ability of a computer system to pause and resume the execution of multiple tasks as needed based on scheduling criteria.

Popular desktop operating systems such as Microsoft Windows and Linux perform preemptive multitasking to allow dozens or even hundreds of simultaneously executing processes to share processor time in a manner that allows all of them to perform their work.

An embedded operating system supporting preemptive multitasking follows a few simple rules to determine which of potentially several tasks is permitted to run each time it performs a scheduling operation.

In embedded systems, a **task** is a distinct thread of execution with a set of associated resources, including processor register contents, a stack, and memory. In a single-processor computer, only one task can be executing at any given time. A **scheduling event** allows the system to select the next task to run and then start or resume its execution. Scheduling events include timer events and task transitions to a blocked state, as well as operating system calls invoked from application code and from **Interrupt Service Routines (ISRs)**.

Tasks in embedded systems are usually in one of three states:

- **Ready state:** The task is prepared to run but it is not actually running because it is in the scheduler's queue awaiting scheduling for execution.
- **Running state:** The task is executing processor instructions.
- **Blocked state:** The task is waiting for an event to occur, such as waiting for a system resource or for the receipt of a signal from a timer.

Each task is assigned a priority by the system developer. Each time a scheduling event occurs, the system identifies the highest priority task that is in either the Ready or Running states and transfers control to that task, or leaves it in the Running state if it is already there. The switch from one task to another involves storing the context information, primarily the processor register contents, associated with the departing task in its **Task Control Block (TCB)** and restoring TCB information for the incoming task to the processor registers before jumping to the next instruction in the incoming task's code. Each context switch takes a small amount of time that subtracts from the time available for task execution.

Figure 3.3 presents the operation of the Motor Control algorithm within a preemptive multitasking RTOS. This system has timer events at 20 ms intervals. At each interval, the Motor Control task enters the Ready state and, because it has the higher priority, it immediately enters the Running state and executes its update, and then returns to the Blocked state:

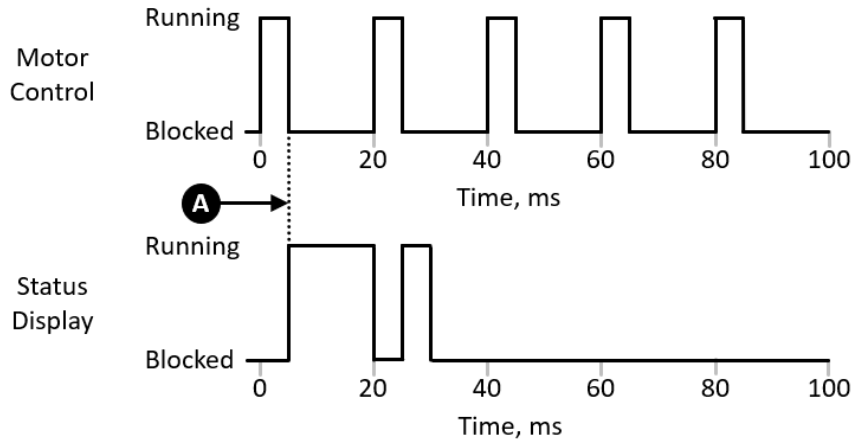


Figure 3.3 – Preemptive multitasking

Every 100 ms, the Status Display task enters the Ready state, but because it is lower priority, the Motor Control task runs first. When the Motor Control task enters the blocked state at time A, the Status Display task enters the Running state and begins execution. At 20 ms, another timer event occurs and the Motor Control task again enters the Ready state. Because it is higher priority, it again runs until it enters the Blocked state. At that point, the Status Display task resumes execution until it completes its update and enters the Blocked state.

The following listing shows an implementation of this system in C using the FreeRTOS RTOS:

```
#include "FreeRTOS.h"
#include "task.h"

void InitializeSystem(void);
void UpdateMotorControl(void);
void UpdateStatusDisplay(void);

static void StatusDisplayTask(void* parameters)
{
```

```
    TickType_t next_wake_time = xTaskGetTickCount();
    for (;;)
    {
        const TickType_t block_time = pdMS_TO_TICKS(100);

        vTaskDelayUntil(&next_wake_time, block_time);

        UpdateStatusDisplay();
    }
}

static void MotorControlTask(void* parameters)
{
    TickType_t next_wake_time = xTaskGetTickCount();
    for (;;)
    {
        const TickType_t block_time = pdMS_TO_TICKS(20);

        vTaskDelayUntil(&next_wake_time, block_time);

        UpdateMotorControl();
    }
}

void main(void)
{
    xTaskCreate(StatusDisplayTask, "StatusDisplay",
               configMINIMAL_STACK_SIZE,
               NULL, (tskIDLE_PRIORITY + 1), NULL);

    xTaskCreate(MotorControlTask, "MotorControl",
               configMINIMAL_STACK_SIZE, NULL,
               (tskIDLE_PRIORITY + 2), NULL);

    InitializeSystem();
}
```

```
vTaskStartScheduler();  
  
// This point is only reached if memory  
// allocation fails during startup  
for (;;) ;  
}
```

This code defines two tasks as C functions: `StatusDisplayTask()` and `MotorControlTask()`. The same functions that implement the application functionality in the earlier example are used here: `InitializeSystem()`, `UpdateStatusDisplay()`, and `UpdateMotorControl()`. The `vTaskDelayUntil()` function performs a precise time delay to ensure the Motor Control task becomes ready to run every 20 ms and the Status Display task becomes ready to run every 100 ms.

Task priorities in FreeRTOS are assigned with lower numerical values representing lower priorities. The lowest priority is the idle task with a priority of 0, represented by the constant `tskIDLE_PRIORITY`. The **idle task** is provided by the system and executes whenever a scheduling event occurs and there is no other task in the Ready state. The Status Display task is assigned a priority one higher than the idle task and the Motor Control task is assigned a priority two higher than the idle task.

This example should make it clear that the use of preemptive multitasking takes a great deal of work off the shoulders of the system developers when working with multiple real-time tasks executing at different update rates. Although this example included only two tasks, the principles of task prioritization and preemptive multitasking support an arbitrary number of tasks in a system, limited only by available system resources and execution time constraints.

While preemptive multitasking relieves system developers from the need to fit code execution within narrow time slots, there is still a limit to how much execution time each task in a multitasking system can consume and remain guaranteed to meet its timing constraints. The next section introduces rate-monotonic scheduling, which provides a method to guarantee that timing constraints will not be violated as long as certain conditions are met.

Rate-monotonic scheduling

The **processor utilization** of a periodic task is the maximum execution time of the task divided by the execution interval of the task, expressed as a percentage. In our example, with the extended Status Display processing time, the utilization of the Motor Control task is $(5 \text{ ms} / 20 \text{ ms}) = 20\%$, and the utilization of the Status Display task is $(20 \text{ ms} / 100 \text{ ms}) = 20\%$. The total processor utilization for this application is thus $20\% + 20\% = 40\%$.

While we can be confident that our two-task system represented in *Figure 3.3* will always satisfy its timing constraints, how can we retain this confidence if we add more tasks to the system, each updating at its own rate, and each with its own processor utilization?

Rate-monotonic Scheduling (RMS) provides an answer to this concern. The timing constraints of a real-time system with periodically scheduled tasks are guaranteed to be met if the following conditions and assumptions are satisfied:

- Task priorities are assigned with the highest priority going to the most frequently executing task, decreasing monotonically down to the lowest priority assigned to the least frequently executing task.
- A task cannot block waiting for a response from another task.
- The time to perform task scheduling and context switching is considered negligible.
- The total processor utilization (the sum of the processor utilizations for all n tasks) is no greater than $n(2^{1/n} - 1)$.

The following table employs this formula to present the RMS processor utilization limits for task counts from 1 to 8:

Number of tasks	Maximum processor utilization
1	100.00%
2	82.84%
3	77.98%
4	75.68%
5	74.35%
6	73.48%
7	72.86%
8	72.41%

In our example, the total processor utilization was 40%. From the preceding table, we see that we can increase the processor utilization as high as **82.84%** with two tasks and still be guaranteed that timing constraints will be satisfied, as long as the RMS criteria are satisfied.

As the number of tasks in a system increases, the maximum processor utilization decreases. As the number of tasks becomes very large, the maximum processor utilization converges to a limit of 69.32%.

The RMS limit on processor utilization is conservative. It may be possible for a system to run at a higher level of processor utilization for a particular number of tasks than is shown in this table.

In addition to preemptive multitasking, most popular RTOS implementations support a variety of standard features, while also requiring developers to remain aware of certain potential problem areas. The next section introduces some standard RTOS capabilities and areas of concern for system architects.

Understanding key RTOS features and challenges

Several standard capabilities are included in most of the RTOS implementations that are in wide use today. Some of these features enable efficient communication among tasks in a manner consistent with real-time operation. While common, not all of the following features are universally available in all RTOSes.

Mutexes

A **mutex**, which stands for **mutual exclusion**, is a mechanism for managing access to a shared resource among tasks. A mutex is conceptually identical to a global variable that can be read and written by all tasks. The variable has the value 1 when the shared resource is free, and 0 when it is in use by a task. When a task needs to gain access to the resource, it reads the variable and, if it is free, with the value 1, sets it to 0 to indicate the mutex is owned by a task. The task is then free to interact with the resource. When the interaction is complete, the task sets the mutex to 1, thereby releasing ownership.

If a task attempts to take ownership of the mutex while the mutex is held by another task, the first task will block until ownership is released by the second task. This remains true even if the task holding the mutex has a lower priority than the task requesting it. *Figure 3.4* presents an example of how mutex ownership can interact with task priorities:

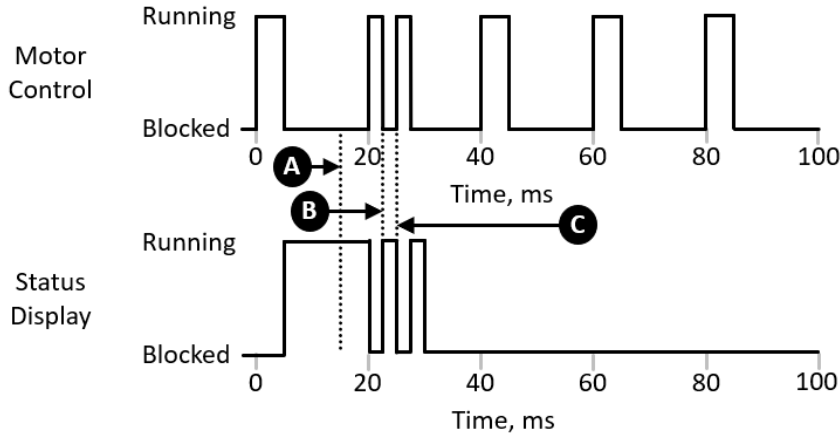


Figure 3.4 – Mutex ownership interaction with task priorities

This diagram shows the two tasks running in the preemptive multitasking environment of *Figure 3.3*, except in this case, the Status Display task takes ownership of a mutex at time **A**. At time 20 ms, the Motor Control task becomes scheduled and begins executing. Midway through processing, at time **B**, the task attempts to take the same mutex, but because the resource is not available, the Motor Control task blocks. This allows the Status Display task to resume processing until it frees the mutex at time **C**. This unblocks the Motor Control task, allowing it to take the mutex and resume execution. The Motor Control runs until it completes its update, and then blocks, waiting for the next cycle. The Status Display task then resumes until it, too, blocks, waiting for its next update.

In this example, the blocking of the Motor Control task resulted in a delay in the completion of its processing, which we have already indicated is unacceptable in the system design. We must also note that this violated one of the RMS criteria, specifically, the admonishment to avoid execution dependencies between tasks. Introducing such complexities does not mean the system will not be able to work properly; it simply means additional analysis and testing will be required to ensure proper system operation under all conditions.

This example demonstrates some of the complexity and pitfalls you may encounter when working with inter-task dependencies. A good rule of thumb when working with mutexes is to hold the mutex for the smallest possible length of time before releasing it.

Semaphores

A **semaphore** is a signaling mechanism that synchronizes operations across tasks. The semaphore is a generalization of the mutex and can be of two types: a binary semaphore or a counting semaphore. A **binary semaphore** functions similar to a mutex, except its purpose is to send a signal to another task. If a task attempts to *take* a semaphore while it is held by another task, the requesting task will block until the task holding the semaphore *gives* it.

A **counting semaphore** contains a counter initialized to an upper limit. Each time a task takes a counting semaphore, the counter decrements by one. When the counter reaches zero, attempts to take the semaphore will block until at least one semaphore holder gives it, which increments the counter.

One application of a semaphore involves the reception and processing of incoming data. If the I/O device associated with the incoming data stream uses a processor interrupt to trigger an ISR, the ISR can retrieve the data from the peripheral device, store it in a memory buffer, and give a semaphore that unblocks a task waiting for incoming data. This design approach allows the ISR to exit as quickly as possible, making the system more responsive to subsequent interrupts and minimizing delays in task execution.

It is generally advisable to spend as little time as possible processing each ISR, which means handing off processing duties to a task via a semaphore is an effective way to reduce the latency of subsequent interrupts. When the task finishes processing the incoming data, it again attempts to take the semaphore, which will block if no additional data has arrived since the last time it took the semaphore.

Queues

A **queue**, sometimes called a **message queue**, is a one-way communication path between tasks. A sending task places data items in the queue and the receiving task removes them in the same order they were inserted. If the receiver attempts to read a queue that is empty, it can choose to block while waiting for data to be placed in the queue.

Similarly, if the queue is full and the sender attempts to place data in the queue, it can choose to block until there is space available. Queues are commonly implemented using a fixed-size memory buffer that can contain an integer number of fixed-size data items.

Event flags

Event flags, also known as **event groups**, are collections of single-bit flags that signal the occurrence of events to tasks. Event flags support a wider range of inter-task communication signaling methods than semaphores. Features of event flags include the following:

- A task can block waiting for a combination of event flags. The task will only become unblocked when all of the events indicated by the selected flags have occurred.
- Multiple tasks can block waiting on a single event flag. When the event occurs, all of the waiting tasks are unblocked. This differs from the behavior of semaphores and queues, which only unblock a single task when the event occurs.

Event flags are useful in specific situations, such as broadcasting a notification that must be received by multiple tasks, or waiting for a combination of activities performed by different tasks to complete.

Timers

Timers provide a different method of scheduling future events than the task scheduling mechanism previously discussed. A **timer** provides a means for scheduling a call to a function at a specified time in the future. The function to be called at that time is an ordinary C language function specified by the developer. This function is identified as the **timer callback function**.

The call to the timer callback function takes place in the context of a system-provided task that obeys the regular rules of task scheduling. In other words, the timer callback function will only be called if, when the specified time arrives, the system task in control of timer function calls is the highest priority task that is ready to run. If a higher-priority task is executing at that time, the call to the timer callback function will be delayed until the higher-priority task blocks. The system developer has the ability to specify the priority of the timer callback scheduling task.

Timers can be configured in one-shot mode or repetitive mode. In one-shot mode, the timer callback function is executed one time after the delay expires. In repetitive mode, the timer callback function executes periodically with a period equal to the timer delay.

Dynamic memory allocation

Like desktop computer operating systems, RTOSes generally provide mechanisms to allocate and release blocks of memory. Consider a word processing program running under Windows or Linux. When the user opens a document file from disk, the program determines the amount of memory needed to hold the entire document, or at least part of it, and requests that amount of memory from the operating system. The program then reads the contents of the document into the newly allocated memory region and allows the user to work with it. As the user edits the document, more memory may be needed to hold additional content. The word processor sends additional allocation requests to the operating system when needed to maintain sufficient space to hold the document content. When the user closes the document, the program writes the updated document to disk and releases the memory it was using for the document data.

Similar actions take place in embedded systems, though instead of working with word processor documents, the system is usually working with sensor input such as temperature measurements, button presses, or streams of audio or video data. For some real-time embedded applications, it makes sense to perform dynamic memory allocation as part of routine system operation. There are, however, some well-known problems that can arise in embedded applications that use dynamic memory allocation.

The C language is widely used in embedded system development. This programming language does not provide automatic allocation and deallocation of memory as objects and data structures are created and destroyed. It is up to the system developer to ensure that the allocation and freeing of memory takes place in a correct, efficient, and reliable manner.

Memory leaks and fragmentation are two types of problems that tend to cause issues when using dynamic memory allocation in real-time embedded systems.

Memory leaks

If the system repetitively performs memory allocation, perhaps to temporarily store blocks of incoming data, the system must eventually release the memory to ensure there will be space available for future incoming data.

The region of system memory used for dynamic allocation is called the **heap**. If allocated memory is not released in a timely manner, the available heap space will eventually become exhausted. If the operation of freeing each memory block after use is either mistakenly left out of the code or bypassed for some reason, or if the memory blocks are retained for such a long time that the available memory is reduced to zero, a **heap overflow** will occur. In this situation, additional attempts to allocate memory will fail.

We can expect the system to crash or exhibit other forms of unintended behavior if a heap overflow occurs in the absence of effective steps to detect the overflow and correct the situation. In the C language, a call to the `malloc()` memory allocation function returns the special value `NULL` when it is unable to allocate the requested size block of memory.

Tutorial examples you may come across demonstrating the use of `malloc()` often assume the call always succeeds, and immediately begin using the return value as a pointer to the freshly allocated block. When `malloc()` fails to allocate the requested block of memory, the return value of `NULL` is, in effect, an address of zero. In a desktop operating system, any attempt to read or write memory at the address zero results in a memory access violation and, normally, the program exits with an error message.

In an embedded system, depending on the particular hardware architecture, it may be perfectly acceptable to read and write address zero. These low addresses usually contain important processor registers, and writing arbitrary data to them (because the code assumed `malloc()` returned a valid pointer to a memory block but it received zero instead) is likely to cause the system to abruptly stop operating correctly. Because this type of error occurs only after the system has been running long enough to consume all available memory, it may be very difficult to identify and debug the source of the problem.

Heap fragmentation

If a real-time application performs dynamic memory allocation, it is possible for the response time performance to be perfectly adequate at system startup and for some time thereafter, but degrade over time. If frequent memory allocation and free operations take place during system operation, even if there is no heap overflow, it is possible, and even likely, that the managed memory region will become fragmented into free blocks of various sizes. When this happens, a memory allocation for a large block might not be immediately possible even though plenty of free memory is available. The memory manager will have to consolidate some number of smaller free blocks into a single block that can be returned for use by the calling code.

In a highly fragmented memory scenario, the process of consolidating multiple blocks can take a long time, which may lead to failure of the system to meet its timing deadlines. Bugs such as this (and it is a bug, even though the system eventually performs in a functionally correct manner) might occur rarely, with serious effects on system behavior, and are often difficult to replicate in a debugging environment.

Deadlock

When using mutexes to manage access to multiple shared resources, it is possible to encounter situations where multiple tasks attempt to take more than one semaphore each and enter a situation where the tasks become permanently blocked. This is called **deadlock**.

For example, assume the Motor Control and Status Display tasks have access to mutexes associated with shared system resources. Assume mutex M_{data} controls access to a data structure shared among tasks and mutex $M_{console}$ controls access to the output channel for writing console messages. *Figure 3.5* presents the timeline for this scenario:

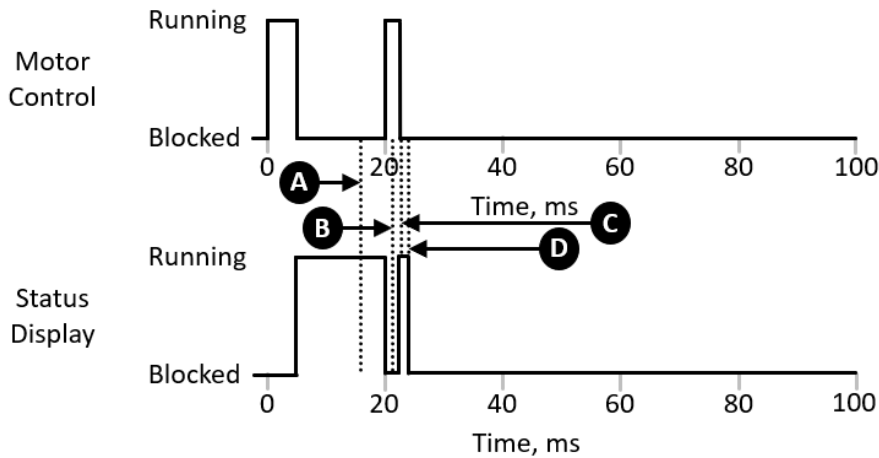


Figure 3.5 – Deadlock example

During its execution, the Status Display task is preparing to write a message to the console. The Status Display task has taken $M_{console}$ at time **A** and is formatting the message to be displayed. The task is interrupted to schedule the higher-priority Motor Control task at the 20 ms mark.

During its processing, the Motor Control task takes M_{data} at time **B** and begins working with the data structure. While working with the structure, it detects an out-of-limits condition within the data and determines it must write a message to the console describing the condition. The Motor Control task then attempts to take $M_{console}$ at time **C** so it can write the message.

Since the Status Display task already has ownership of the $M_{console}$ mutex, the Motor Control task blocks and the Status Display task resumes execution, preparing its own message for display on the console. To populate the message, the Status Display task must gather some information from the shared data structure, so it attempts to take M_{data} at time **D**.

At this point, both of the tasks are stuck, with no way out. Each task has taken one of the two mutexes and is waiting for the other mutex to become free, which cannot happen because both tasks are blocked.

In this example, the actions taken by each task, viewed in isolation, appear reasonable, but when they interact through the mutexes, the result is an immediate halt to system operation, at least for the affected pair of tasks. This represents a catastrophic failure in terms of system performance.

Avoiding the possibility of deadlock is a responsibility of the system architect. There are a couple of rules of thumb that will ensure deadlock cannot occur in a system design:

- Whenever possible, avoid locking more than one mutex at a time.
- If you must lock multiple mutexes in multiple tasks, ensure they are locked in the same sequence in each task.

Some RTOS implementations can detect the occurrence of a deadlock and return an error code when attempting to take a semaphore that would result in deadlock. The algorithm required to implement this capability is considered expensive in terms of embedded resources (specifically, code size and execution time), and avoiding the possibility of deadlock through careful system design is often the superior approach.

Priority inversion

A situation that causes a violation of task prioritization can occur when tasks of varying priorities use a mutex to control access to a shared resource. This situation can occur with three tasks of different priorities.

Let's add another task to our system for performing measurements with an ultrasonic sensor. This task runs at 50 ms intervals and takes up to 15 ms to complete each execution cycle. To comply with the requirements of RMS, this task must have a priority between those of the Motor Control task, which runs at 20 ms intervals, and the Status Display task, which runs at 100 ms intervals.

We can quickly check whether the system remains schedulable under the RMS criteria. In the *Rate-monotonic scheduling* section, we saw that the total processor utilization for the two-task application is 40%. The new task consumes another $(15 \text{ ms} / 50 \text{ ms}) = 30\%$ of processor time, for a combined total utilization of 70%. From the table in the *Rate-monotonic scheduling* section, we see that the RMS schedulability threshold for a three-task system is 77.98%. Because our processor utilization is below the threshold, we can be certain that as long as the RMS criteria are met, the system will meet timing deadlines.

Let's say the new Sensor Input task is first scheduled at time 10 ms and again at 60 ms. Because the Motor Control task is also scheduled at 60 ms, the Sensor Input update must block until the Motor Control task update is complete. We will assume that this deviation from precise periodic update intervals is not a significant issue for the application. This execution timing sequence is shown in *Figure 3.6*:

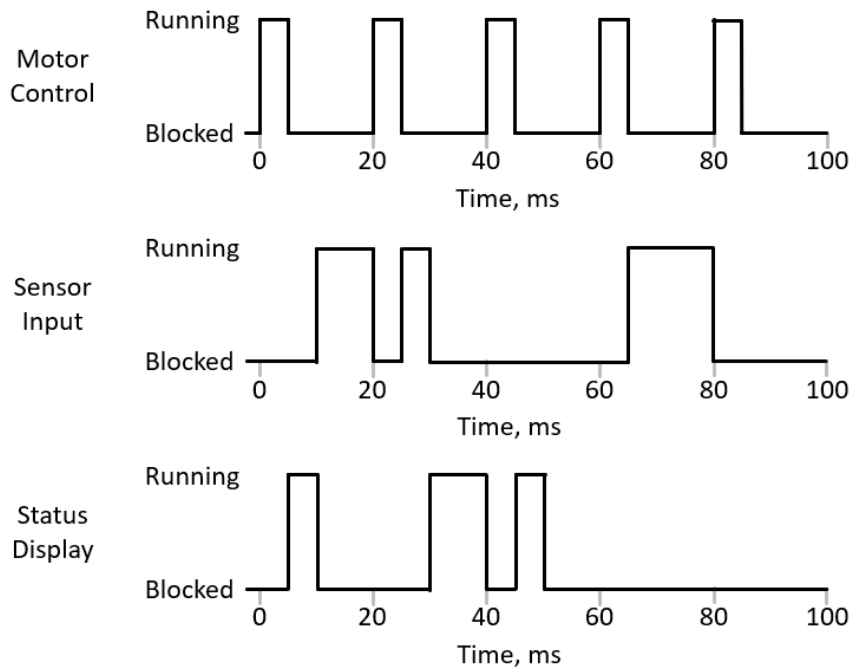


Figure 3.6 – Three-task execution sequence

The Status Display task update is now broken into three separate execution time segments. While this may appear unusual, such behavior is perfectly normal in a preemptive multitasking system.

Let's introduce an innocuous-seeming dependency between the Status Display task and the Motor Control task. We learned from our problematic implementation of mutex usage in *Figure 3.4* that we need to limit the length of time a mutex is held by a lower-priority task to the absolute minimum. In the three-task system, the Status Display task now only holds the mutex protecting the shared data structure long enough to copy the data it needs before releasing the mutex. We expect this to substantially reduce, though not entirely eliminate, the unacceptable Motor Control task execution delay of *Figure 3.4*.

Unfortunately, when we run the system, we see the timing response is occasionally much worse, as shown in *Figure 3.7*:

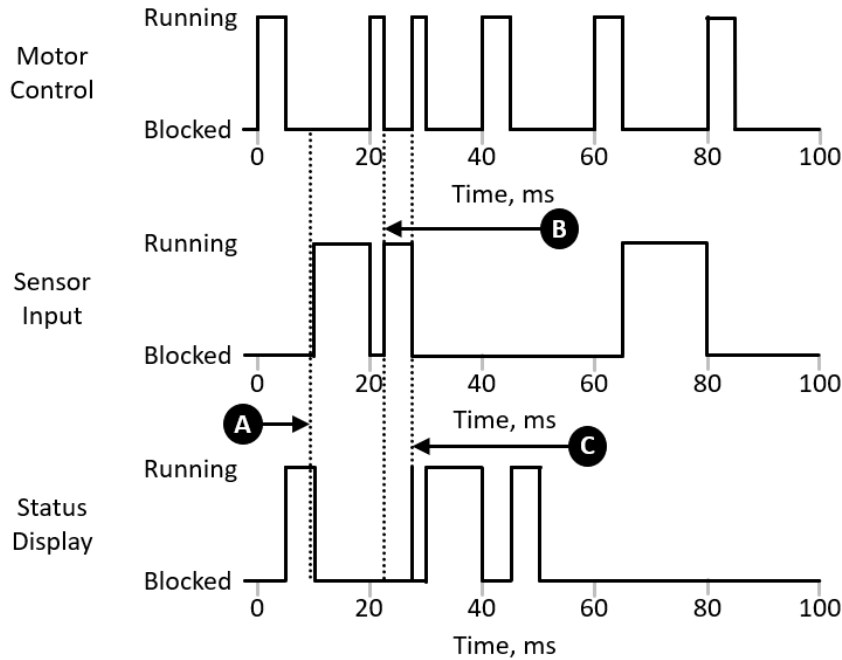


Figure 3.7 – Priority inversion example

What's happening here? At time **A**, the Status Display task takes the mutex. Even though it intends to release it very shortly, the Sensor Input task becomes ready to run and begins execution before the Status Display task can release the mutex. At time 20 ms, the Motor Control task is scheduled and begins execution.

At time **B**, the Motor Control task attempts to take the mutex, which causes it to block. The Sensor Input task is ready to run, so it resumes execution at this point. It is not until the Sensor Input task finishes its update and blocks that the Status Display task becomes ready to run again, at time **C**. When the Status Display task resumes, it quickly finishes reading the data structure and releases the mutex. This finally allows the Motor Control task to take the mutex and finish its (much delayed) execution.

The problem here was that the mid-priority task (Sensor Input) was able to run even though the higher-priority task (Motor Control) would have been ready to run had the low priority task (Status Display) been allowed to continue execution and release the mutex. This situation is called **priority inversion**.

The standard RTOS solution to the priority inversion problem is to implement priority inheritance. In **priority inheritance**, whenever a higher-priority task blocks waiting for the release of a resource held by a lower-priority task, the lower-priority task temporarily raises its priority to that of the higher-priority task. Once the resource has been freed by the lower-priority task, that task returns to its original priority.

Figure 3.8 shows the same situation as Figure 3.7, except priority inheritance is now implemented:

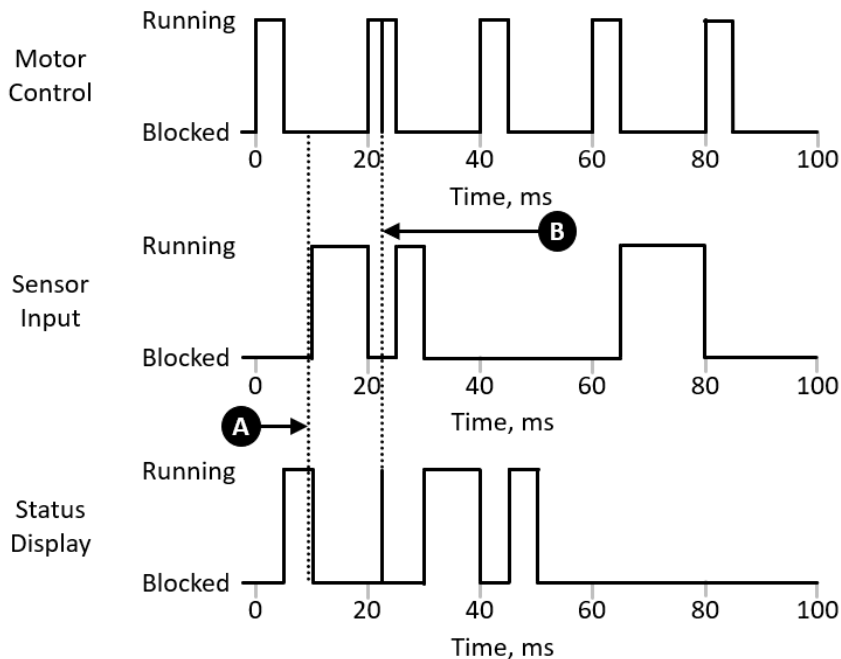


Figure 3.8 – Priority inheritance

In this diagram, the Status Display task again takes the mutex at time A. At time B, the Motor Control task attempts to take the mutex. The system elevates the priority of the Status Display task to that of the Motor Control task, ensuring the Sensor Input task does not run. This gives the Status Display task an opportunity to quickly complete reading the structure and release the mutex. The Motor Control execution timeliness is now significantly improved in comparison to Figure 3.7.

The next section briefly introduces some popular RTOSes and highlights their features and the categories of real-time embedded applications best suited to each of them.

Popular real-time operating systems

When selecting an RTOS for a particular real-time embedded system architecture and application domain, it is important to consider a variety of technical and non-technical factors in the selection process. Almost all popular RTOSes support prioritized preemptive multitasking, mutexes, semaphores, queues, event flags, timers, and dynamic memory allocation. All of the RTOSes listed in this section include these features.

Some key technical attributes that differentiate among the various RTOSes are as follows:

- **Feature richness:** Some RTOSes are intended to be as small as possible, consuming the absolute minimum quantity of ROM, RAM, and processor cycles in tiny microcontrollers. Other RTOSes are designed to support a large number of tasks and complex protocol stacks such as TCP/IP running on a 32-bit processor.
- **Memory protection and virtual memory management:** Simple microcontrollers and low-end microprocessors typically support only direct physical addressing of ROM and RAM. Many mid-range processors provide a mechanism for controlling memory access called a **Memory Protection Unit (MPU)**. With the use of MPU functionality, memory regions can be isolated and protected to ensure critical system functions continue running even if less critical tasks experience problems that cause them to erroneously access memory and, perhaps, crash. At a more sophisticated level, 32-bit processors often include a **Memory Management Unit (MMU)**, providing each running process with its own protected virtual address space. RTOSes supporting virtual memory take advantage of MMU hardware to encapsulate each process (which is conceptually similar to a task) in its own dedicated memory region so that tasks cannot interact with each other, intentionally or otherwise, except through system-provided communication channels.
- **Modularity and configurability:** Adding features to an RTOS increases the amount of ROM required for code and RAM required for data. Most RTOSes provide configuration options to include only those features that an application actually needs in the compiled memory image, reducing the amount of memory and processing time required.
- **Processor architecture support:** RTOSes generally come with a list of processor architectures and specific processor models supported by the implementation. These processor-specific implementations generally come with a code library called a **Board Support Package (BSP)**. A BSP includes an implementation of the RTOS tailored to a specific processor model and, often, to a particular circuit board and its I/O interfaces. The BSP also includes a library of device drivers that enables the system developer to begin implementing an application using a standard programming interface to the processor hardware. If you have already selected a processor architecture for your application, this will constrain which RTOSes are suitable for your use.

- **Supporting tools and accessories:** In addition to the core RTOS and associated device drivers, you may require additional hardware and software tools to support the development process, such as debuggers, execution tracers, timing analyzers, and memory usage analyzers. Support for such tools varies among the available RTOSes.

Some non-technical attributes that you may wish to consider during RTOS selection are as follows:

- **Choosing commercial or open source:** Paying a license fee for a commercial RTOS provides some significant benefits, including technical support and some promise of future RTOS sustainment. Of course, it also costs money. There are many free-to-use RTOS implementations available as well, but each comes with its own licensing requirements, community of users, and prospects for future support.
- **Vendor lock:** Once you implement your application using a particular RTOS, you are, to some degree, committed to continued use of that RTOS. If the commercial RTOS vendor you select goes out of business or changes its licensing terms in an undesirable manner, or if the open source RTOS you choose falls out of favor and becomes unmaintained, you may have to make a choice to perform a potentially painful re-architecting of your design.
- **Formal certification:** For safety-critical applications, such as in aircraft, automobiles, and medical devices, some RTOSes have received formal certification as suitable for use in those contexts. If you are building a system where such a certification is important, this will focus your search on the RTOSes that have achieved the appropriate certification.
- **Software license terms:** A license for a commercial RTOS contains whatever terms the vendor chooses to put in their license agreement. Open source RTOSes are commonly licensed under one of the MIT, Apache, or GPL licenses. The MIT and Apache licenses are considered permissive, meaning developers can take the software and use it for their own purposes, including commercial applications, without being compelled to make their own source code public. The GPL, on the other hand, requires developers who incorporate GPL code into a product they distribute to make their code available to all who request it. This is obviously a highly simplified description of the distinction between these licenses. Many factors can combine to make licensing issues for products based on open source code extremely complex.

The following sections briefly describe a number of popular RTOSes and highlight the unique features of each. The RTOSes are listed in alphabetical order to avoid implying a preference for any particular one. This list is not intended to be exhaustive.

embOS

embOS is a commercial RTOS produced by SEGGER Microcontroller LLC. **embOS** is intended for use across a wide range of real-time applications, from single-chip, battery-powered devices to sophisticated systems running on advanced processors. embOS supports virtually all embedded processor architectures from major vendors as well as a wide variety of compilers for those architectures.

An edition of embOS is available with full MPU support. A separate edition is safety certified to the IEC 61508 SIL 3 standard, which certifies a safety-focused RTOS software development process, and IEC 62304 Class C, which represents suitability for use in medical device applications.

A free version of embOS is available for non-commercial use. This version does not include embOS source code. For commercial use, or to receive source code, you must purchase a license. See <https://www.segger.com/products/rtos/embos/> for more information.

FreeRTOS

FreeRTOS is a free RTOS microkernel developed by Real Time Engineers Ltd. A **microkernel** contains a minimal amount of code that implements the basic functionality of an RTOS, including task management and inter-task communication.

FreeRTOS provides several options for dynamic memory management, from no memory allocation capability at all to support for the unrestricted allocation and freeing of arbitrarily sized memory blocks. FreeRTOS supports 35 different microcontroller platforms and is written in the C language with a few assembly language functions to support preemptive multitasking.

Amazon maintains an extended version of FreeRTOS named a:FreeRTOS. This version includes libraries that provide IoT capabilities specifically focused on working with Amazon Web Services. A version of FreeRTOS named SAFERTOS, certified to the IEC 61508 SIL 3 standard, is intended for safety-critical applications.

FreeRTOS is made available under the MIT license. For system developers who prefer a commercially licensed RTOS, OPENRTOS is a commercially licensed variant of the Amazon a:FreeRTOS. See <https://www.freertos.org/> for more information.

Example applications in future chapters will use FreeRTOS because of its free nature, permissive licensing, and the fact that it comes pre-integrated in the Xilinx tool suite.

INTEGRITY

The INTEGRITY RTOS from Green Hills Software is targeted at applications with the highest requirements in terms of safety, security, and reliability. **INTEGRITY** provides a variety of middleware options for functions such as TCP/IP communication, web services, and 3D graphics. INTEGRITY is targeted at applications in the automotive, aviation, industrial, and medical domains.

INTEGRITY has been safety certified in a variety of application areas, including aviation applications, high security applications, medical devices, railway operations, industrial control, and automotive applications. INTEGRITY provides a secure virtualization infrastructure as well as support for multicore processors. This RTOS is supported on a wide range of higher-end microprocessor architectures.

INTEGRITY is commercially licensed. There does not appear to be a free version available. See <https://www.ghs.com/products/rtos/integrity.html> for more information.

Neutrino

The QNX Neutrino RTOS from BlackBerry is intended to provide performance, safety, and security in critical applications. **Neutrino** is intended for applications in the automotive, medical, robotics, and industrial domains and is built with a microkernel architecture that isolates drivers and applications so that the failure of one component does not bring down the entire system.

Neutrino supports ARMv7, ARMv8, and x86-64 processors and SoCs. The RTOS includes a variety of networking and connectivity protocols, including TCP/IP, Wi-Fi, and USB.

Neutrino is commercially licensed. A free evaluation version is available. See <https://blackberry.qnx.com/en/software-solutions/embedded-software/qnx-neutrino-rtos> for more information.

μc/OS-III

μc/OS-III is a free RTOS focused on reliability and performance from Micrium, which is part of Silicon Labs. μc/OS-III includes support for TCP/IP, USB, CAN bus, and Modbus. It also has a GUI library that supports the development of smartphone-like graphics displays on touchscreen devices. μc/OS-III is written entirely in ANSI C. This RTOS runs on an extensive range of processor architectures.

µC/OS-III is safety certified for use in aviation, medical, transportation, and nuclear systems. µC/OS-III is released under the Apache license. See <https://www.micrium.com/rtos/> for more information.

VxWorks

VxWorks is a commercially licensed 32- and 64-bit RTOS from Wind River Systems. **VxWorks** is targeted at applications in the aerospace, defense, medical, industrial, automotive, IoT, and consumer electronics domains. Supported architectures include POWER, ARM, Intel, and RISC-V. VxWorks supports multicore processors and hypervisor implementations.

A safety-certified edition is available for use in aviation, automotive, and industrial applications. A VxWorks edition is available that supports architectural partitioning for aviation applications in a manner that permits modification of components in one partition with a requirement to only recertify that partition and not the entire system.

Important note

The Mars Pathfinder spacecraft that landed on the Red Planet on July 4, 1997 used VxWorks as its RTOS. During its first few days on the surface, the spacecraft began to experience full system resets, resulting in the loss of collected data. The root cause of this problem was traced to a classic priority inversion, much like that of Figure 3.7. Instead of delaying a Motor Control update, the delay of Pathfinder's higher-priority task resulted in the expiration of a watchdog timer, which triggered the system resets. Engineers were able to replicate the problem on an identical system on Earth. The solution: modify a parameter value to turn on priority inheritance for the mutex associated with the problem. Uploading this fix to the spacecraft enabled it to resume normal operation.

VxWorks includes a full suite of development, debugging, and tracing tools. See <https://www.windriver.com/products/vxworks/> for more information.

Summary

This chapter described the methods RTOSes use to ensure real-time responses to inputs. Key features available in common RTOSes were introduced, along with some challenges that commonly arise when implementing multitasking real-time applications. The chapter concluded with a listing of the key features of some popular open source and commercial RTOS implementations.

Having completed this chapter, you now understand what it means for a system to operate in real time and understand the key attributes a real-time embedded system must exhibit. You understand the RTOS features that embedded systems rely upon, as well as some challenges that frequently occur in real-time embedded system designs. You are also familiar with the key features of several popular RTOS implementations.

The next chapter introduces the concepts involved in the design of real-time embedded systems using FPGAs and works through a simple FPGA application example.

Section 2: Designing and Constructing High- Performance Embedded Systems

In this part, you will become familiar with the capabilities of **Field Programmable Gate Arrays (FPGAs)** and learn how to design and construct high-performance circuits based on these devices.

This part of the book comprises the following chapters:

- *Chapter 4, Developing Your First FPGA Program*
- *Chapter 5, Implementing Systems with FPGAs*
- *Chapter 6, Designing Circuits with KiCad*
- *Chapter 7, Building High-Performance Digital Devices*

4

Developing Your First FPGA Program

This chapter begins with a discussion on the effective use of FPGA devices in real-time embedded systems and continues with a description of the functional elements contained within standard FPGAs. The range of FPGA design languages, including **Hardware Description Languages (HDLs)**, block diagram methods, and popular software programming languages including C and C++, is introduced. The chapter continues with an overview of the FPGA development process and concludes with a complete example of an FPGA development cycle starting with a statement of system requirements and ending with a functional system implemented in a low-cost FPGA development board.

After completing this chapter, you will know how FPGAs can be applied in real-time embedded system architectures and will understand the components that make up an FPGA integrated circuit. You will have learned about the programming languages used in the design of FPGA algorithms and will understand the sequence of steps to develop an FPGA-based application. You will also have worked through a complete FPGA development example on a low-cost development board using free FPGA software tools.

We will cover the following topics in this chapter:

- Using FPGAs in real-time embedded system designs
- FPGA implementation languages

- The FPGA development process
- Developing your first FPGA project

Technical requirements

The files for this chapter are available at <https://github.com/PacktPublishing/Architecting-High-Performance-Embedded-Systems>.

Using FPGAs in real-time embedded system designs

As we saw in the *Elements of FPGAs* section of *Chapter 1, Architecting High-Performance Embedded Systems*, a typical FPGA device contains a large number of lookup tables, flip-flops, block RAM elements, DSP slices, and other components. While it can be instructive to understand the detailed capabilities of each of these components, such concerns are not necessarily informative during the FPGA development process. The most important constraint to keep in mind is that a specific FPGA part number contains a finite number of each of these elements, and a design cannot exceed those limits when targeted at that particular FPGA model.

Instead, it is more productive to view the FPGA development process from the perspective of the embedded system's statement of requirements. You can begin to develop the FPGA design targeted at a somewhat arbitrarily chosen FPGA model. As development proceeds, you may reach a resource limit or identify an FPGA feature the design requires that is not present in the currently targeted FPGA. At that point, you can select a different, more capable, target and continue development.

Alternatively, as development of the design nears completion, you may realize the target FPGA you originally selected contains excessive resources and the design could be improved by selecting a smaller FPGA, with potential benefits in terms of lower cost, fewer pins, smaller package size, and reduced power consumption.

In either of these situations, it is generally straightforward to switch the targeted FPGA to a different model within the same family. The development tools and design artifacts you have created to this point should be fully reusable with the newly targeted FPGA model. If it becomes necessary to switch to a different family of FPGAs from the same vendor, or to a model from a different vendor, the switchover will likely involve more work.

The point of this discussion is to emphasize that it is not too important to identify a specific FPGA model at the outset of a high-performance embedded system development effort. Instead, early considerations should focus on validating the decision to use an FPGA as part of the design, then, if the FPGA is the best design approach, proceed with the selection of a suitable FPGA vendor and device family.

Example projects in this book will be based on the Xilinx Vivado family of FPGA development tools. Although a Vivado license must be purchased to develop for some Xilinx FPGA families, the FPGA devices in the Artix-7 we will be working with are supported by Vivado for free. The Artix-7 FPGA family combines the attributes of high performance, low power consumption, and reduced total system cost. Similar FPGA device families and development tool suites are available from other FPGA vendors.

FPGA development is a fairly involved process, with a variety of types of analysis and design data input required. To avoid discussing these topics at too abstract a level, and to present concrete results in terms of working example projects, we will be using Vivado throughout the book. Once you are familiar with the tools and techniques discussed here, you should be able to apply them using similar tools from other vendors.

The following sections will discuss some key differentiating features of the families of FPGAs and individual models within those families, including the quantity of block RAM, the quantity and types of I/O signals available, specialized on-chip hardware resources, and the inclusion of one or more hardware processor cores in the FPGA package.

Block RAM and distributed RAM

Block RAM is used to implement regions of memory within an FPGA. A particular memory region is specified in terms of the width in bits (typically 8 or 16 bits) and the depth, which defines the number of storage locations in the memory region.

The total quantity of block RAM in an FPGA is usually specified in terms of **kilobits (Kb)**. The amount of block RAM available varies across FPGA families and among the models within a particular family. As you would expect, larger, more expensive parts generally have a greater quantity of resources that can be used as block RAM.

In Xilinx FPGAs, and to varying degrees in FPGAs from other vendors, a distinct category of memory called distributed RAM is available in addition to block RAM. **Distributed RAM** is constructed from the logic elements used in lookup tables and repurposes the circuitry of those devices to form tiny segments of RAM, each containing 16 bits. These segments can be aggregated to form larger memory blocks when necessary.

Block RAM tends to be used for purposes traditionally associated with RAM, such as implementing processor cache memory or as a storage buffer for I/O data. Distributed RAM might be used for purposes such as the temporary storage of intermediate computation results. Because distributed RAM is based on lookup table circuitry, the use of distributed RAM in a design reduces the resources available for implementing logic operations.

Block RAM can have a single port or dual ports. Single-port block RAM represents the common usage pattern of a processor that reads and writes RAM during operation. Dual-port block RAM provides two read/write ports, both of which can be actively reading or writing the same memory region simultaneously.

Dual-port block RAM is ideal for situations where data is being transferred between portions of an FPGA running at differing clock speeds. For example, an I/O subsystem might have a clock speed in the hundreds of MHz as it receives an incoming data stream. The I/O subsystem writes incoming data to the block RAM as it arrives through one of the FPGA's high-speed I/O channels. A separate subsystem with the FPGA, running at a different clock speed, can read data from the block RAM's second port without interfering with the operation of the I/O subsystem.

Block RAM can also operate in **first-in-first-out (FIFO)** mode. In the example of the incoming serial data stream, the I/O subsystem can insert data words into the FIFO as they arrive and the processing subsystem can read them out in the same order. Block RAM in FIFO mode provides signals indicating whether the FIFO is full, empty, almost full, or almost empty. The definitions of *almost full* and *almost empty* are up to the system designer. If you assign *almost empty* to mean less than 16 items are left in the FIFO, you can then be assured that any time the FIFO does not indicate it is almost empty, you can read 16 items without further checks of data availability.

When using block RAM in FIFO mode, it is vital that the logic inserting items into the FIFO never attempts to write when the FIFO is full, and the logic reading from the FIFO never attempts to read when the FIFO is empty. If either of these events occurs, the system will either lose data or will attempt to process undefined data.

FPGA I/O pins and associated features

Because FPGAs are intended for use in high-performance applications, their I/O pins are generally capable of implementing a variety of high-speed I/O standards. During the implementation of a design with an FPGA development tool suite, the system developer must perform tasks that include assigning functions to particular pins on the FPGA package and configuring each of those pins to operate with the appropriate interface standard. Additional steps must be performed to associate input and output signals within the FPGA model code with the correct package pins.

At the pin level, individual I/O signals are either single-ended or differential.

A **single-ended signal** is referenced to ground. Traditional **Transistor-Transistor Logic (TTL)** and **Complementary Metal Oxide Semiconductor (CMOS)** digital signals operate over a range of 0-5 VDC relative to ground.

Modern FPGAs typically do not support the legacy 5 VDC signal range, but instead support TTL and CMOS signals operating over a reduced voltage range, thereby reducing power consumption and improving speed. **Low Voltage TTL (LVTTL)** signals operate over a range of 0-3.3VDC. **Low Voltage CMOS (LVCMOS)** signals are selectable with signaling voltages of 1.2, 1.5, 1.8, 2.5, and 3.3 V. These signal types are named LVCMOS12, LVCMOS15, LVCMOS18, LVCMOS25, and LVCMOS33. Other high-performance single-ended signal types are available, including **High-Speed Transceiver Logic (HSTL)** and **Stub-Series Terminated Logic (SSTL)**.

Single-ended signals are widely used for low-frequency purposes, such as reading pushbutton inputs and lighting LEDs. Single-ended signals are also used in many lower-speed communication protocols such as I2C and SPI. An important drawback of single-ended signals is that any noise coupled into the wires and printed circuit board traces carrying the signal has the potential to corrupt the input to the receiver. This problem can be substantially reduced through the use of differential signaling.

For the highest data transfer rates, differential signaling is the preferred approach.

Differential signals use a pair of I/O pins and drive opposing signals onto the two pins. In other words, one pin is driven to a higher voltage and the other pin to a lower voltage to represent a 0 data bit and the pin voltages are reversed to represent a 1 bit. The differential receiver subtracts the two signals to determine whether the data bit is 0 or 1. Because the two wires or traces carrying the differential signal are physically located very close together, any noise that couples into one of the signals will couple to the other one in a very similar manner. The subtraction operation removes the vast majority of the noise, enabling reliable operation at much higher data transfer rates than single-ended signals.

A number of differential signal standards are supported by standard FPGAs. Several differential versions of HSTL and SSTL are defined, with a variety of signaling voltage levels for each.

Low-Voltage Differential Signaling (LVDS) was introduced as a standard in 1994 and continues to be used in a variety of applications. An LVDS signaling transmitter produces a constant current of 3.5 mA and switches the direction of the current flowing through the resistor at the receiver to produce state changes representing 0 and 1 data values as shown in *Figure 4.1*:

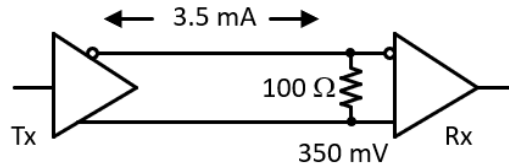


Figure 4.1 – LVDS interface

In LVDS communication, as in the other differential and single-ended signaling standards, it is important for the impedance of the communication path between the transmitter and receiver to closely match the termination impedance, which is 100 Instance 8 in the case of LVDS. If the impedance of the communication channel does not match the termination impedance, reflections can occur on the line, preventing reliable data reception.

The impedance of differential signal trace pairs is a function of the geometry of the pair traces and their relationship to the ground plane. As we will see in *Chapter 6, Designing Circuits with KiCad*, it is straightforward to design circuit boards that satisfy the requirements of high-speed differential signaling standards.

Specialized hardware resources

FPGAs generally include a selection of dedicated hardware resources for functions that are commonly required and are either more efficiently implemented in hardware rather than using synthesized FPGA functions, or not possible to implement with FPGA components. Some examples of these resources are as follows:

- Interfaces to external **dynamic RAM (DRAM)** for storing large quantities of data. These interfaces generally support a common DRAM standard such as DDR3.
- Analog-to-digital converters.
- Phase-locked loops, used for generating multiple clock frequencies.
- Digital signal processing **multiply-accumulate (MAC)** hardware.

These hardware resources enable the development of complex systems with wide-ranging capabilities. Dedicated hardware is provided for functions like the MAC operation because the hardware performance is significantly better than the synthesized equivalent functionality using FPGA logic resources.

Processor cores

Some FPGA families include hardware processor cores for the purpose of combining peak software execution speed with the performance advantages of FPGA-implemented algorithms. For example, the Xilinx Zynq-7000 family integrates a hardware ARM Cortex-A9 processor together with a traditional FPGA fabric.

FPGA designs that do not require a hardware processor can implement a processor using the FPGA resources, referred to as a **soft processor**. Soft processors are highly configurable, though they are generally not capable of matching the performance of a processor implemented in hardware.

The next section will introduce the primary programming languages and data entry methods used to develop FPGA algorithms.

FPGA implementation languages

Implementing a design for an FPGA ultimately comes down to using one or more software-programming-like languages to define the functionality of the device. The traditional languages used for FPGA development are VHDL and Verilog. Current-generation FPGA development tools generally support both of these languages together with the ability to define system configurations using block diagramming techniques. Some tool suites also support the definition of FPGA functionality using the traditional C and C++ programming languages.

VHDL

VHSIC Hardware Description Language (VHDL), where VHSIC stands for *Very High-Speed Integrated Circuit*, has syntax reminiscent of the Ada programming language. VHDL was developed under the guidance of the US Department of Defense beginning in 1983.

Like Ada, VHDL tends to be quite verbose and rigidly structured. In programming language terms, VHDL is strongly typed. The language contains a predefined set of base data types, principally `boolean`, `bit`, `bit_vector`, `character`, `string`, `integer`, `real`, `time`, and `array`. All other data types are defined in terms of the base types.

A set of VHDL libraries has been defined by the **Institute of Electrical and Electronics Engineers (IEEE)** and formalized as the IEEE 1164 standard, *Multivalued Logic System for VHDL Model Interoperability*. These libraries define the set of logic values to be used in the VHDL language. This library includes a type named `std_logic`, which represents, a 1-bit signal. The logical values within the `std_logic` type are represented by the character literals shown in the following table:

Literal	Value
U	Uninitialized
X	Strong drive, unknown logic value
0	Strong drive, logic zero
1	Strong drive, logic one
Z	High impedance
W	Weak drive, unknown logic value
L	Weak drive, logic zero
H	Weak drive, logic one
-	Don't care

The "strong" 0 and 1 values in the preceding figure represent signals driven to the specified binary state. The "weak" signals represent signals driven on a bus with multiple drivers where any driver can assert itself on the bus, overriding the other drivers. The Z value represents a CMOS output in the high-impedance state, where rather than driving the bus to a 0 or 1 state, the output is instead effectively disconnected from the bus and does not drive it at all. The U state represents the default values for all signals. When performing circuit simulation, any signal in the U state will be detected, which likely indicates an uninitialized value is being used unintentionally. The X state is associated with wires that do not have any outputs driving them. The – state represents inputs that are unused, and therefore it does not matter what state they are in.

VHDL circuit designs generally begin by importing the IEEE 1164 libraries via the following statements:

```
library IEEE;
use IEEE.std_logic_1164.all;
```

We will use VHDL in our project example later in the chapter. This is not intended to represent a strong preference for VHDL over Verilog. Both hardware definition languages are fully capable of representing essentially any design that can be synthesized for an FPGA.

Verilog

The Verilog **Hardware Description Language (HDL)** was introduced in 1984 and became standardized as IEEE 1364 in 2005. In 2009, the Verilog standard was combined with the *SystemVerilog* standard to produce IEEE Standard 1800-2009. SystemVerilog contains extensive facilities for performing system verification, in addition to the hardware design features present in Verilog.

Verilog was designed to resemble the C programming language, including similar operator precedence and the use of some of the same control flow keywords, including `if`, `else`, `for`, and `while`.

Verilog uses the concept of a *wire* to represent signal states. A signal value can take any of the values 0, 1, don't care (x), or high impedance (z), and can have a *strong* or *weak* signal strength.

Both VHDL and Verilog define language subsets that can be used to design logic circuitry. These subsets are referred to as the **synthesizable** language subsets. Additional language features beyond the synthesizable subsets are available to support tasks such as circuit simulation. We'll see an example of this later in this chapter.

Non-synthesizable language constructs tend to behave more like traditional software programming languages. For example, a non-synthesizable `for` loop iterates through a block of code sequentially the specified number of times, just like in a regular programming language. A synthesizable `for` loop, on the other hand, becomes effectively unrolled to generate a collection of replicated hardware constructs that execute in parallel representing each iteration of the loop.

Block diagrams

At a level of abstraction above the text-based HDLs, modern FPGA development tool suites support the rapid configuration of system designs incorporating complex logic components such as microprocessors and sophisticated I/O devices using a block structure format. *Figure 4.2* is an example of a portion of a block diagram for a Xilinx FPGA design incorporating a MicroBlaze soft processor:

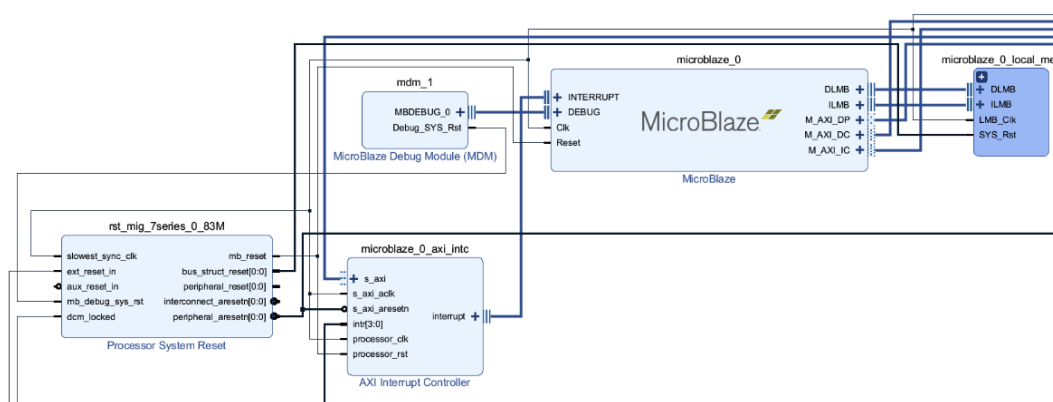


Figure 4.2 – Block diagram containing a MicroBlaze soft microprocessor

The **MicroBlaze processor** is a processor core provided with the Xilinx Vivado tool suite for use in FPGA designs in processor families including the Artix-7.

While the use of block diagrams provides a visually intuitive way to organize the instantiation and interconnection of complex logic elements in an FPGA design, it is important to remember that behind the diagram, the development tool generates VHDL or Verilog code to define the components and their connections. The block diagram is simply a user interface for managing the configuration of these components.

After you develop a block diagram, you can examine the generated HDL code, which will be contained in files associated with the project. In the diagram of *Figure 4.2*, a file named `design_1_microblaze_0_0_stub.vhdl` is produced from the diagram. This file begins with the following VHDL code:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity design_1_microblaze_0_0 is
  Port (
    Clk : in STD_LOGIC;
```

```
Reset : in STD_LOGIC;  
Interrupt : in STD_LOGIC;  
Interrupt_Address : in STD_LOGIC_VECTOR ( 0 to 31 );  
Interrupt_Ack : out STD_LOGIC_VECTOR ( 0 to 1 );  
Instr_Addr : out STD_LOGIC_VECTOR ( 0 to 31 );  
Instr : in STD_LOGIC_VECTOR ( 0 to 31 );
```

This code begins with a reference to the IEEE 1164 standard library, then defines an interface to the MicroBlaze processor that exposes the signals you would expect on a microprocessor, including the system clock, reset, interrupt request, and interrupt vector inputs; interrupt acknowledge and instruction address outputs; and a bus for the instructions retrieved from memory.

This code makes use of the IEEE 1164 library data types for single-bit signals (STD_LOGIC) and for multi-bit bus signals (STD_LOGIC_VECTOR).

The code in the listing defines the interface to the MicroBlaze processor, but it does not contain the HDL definition of the processor itself. Complex HDL designs for components such as microprocessors are considered valuable **Intellectual Property (IP)** and the commercial entities that develop these designs often take steps to ensure they are not used without appropriate licensing. When vendors distribute IP for use by their customers, it may be provided in a compiled format that is opaque to end users. This allows users to incorporate the IP into their designs, but they cannot examine the HDL used to develop it. This is conceptually similar to software developers who release a library in compiled form but do not provide the source code.

C/C++

A number of vendors offer software tools that translate traditional high-level programming languages, often C and C++, into HDL code for use in FPGA development. This approach may be attractive if you have a complex algorithm in C or C++ that you would like to accelerate in an FPGA implementation. These tools also allow software developers who are familiar with C/C++ to immediately begin implementing code for FPGA use without a need to learn one of the HDLs.

While FPGA development tools for these high-level languages are capable of significant optimization of the resulting FPGA implementation of the C/C++ code algorithm, there is still something of a disconnect in that the C/C++ execution model involves the sequential execution of statements while the native FPGA environment consists of parallel hardware components. The FPGA design resulting from C/C++ code typically resembles a collection of state machines that manage the sequential execution of the operations defined in the programming language statements. Depending on the availability of opportunities for parallel execution within the C/C++ code, an FPGA implementation may provide a significant performance enhancement compared to running the same code on a traditional processor.

In modern FPGA development tool suites, all of the methods of FPGA implementation described in this section (VHDL, Verilog, block diagrams, and C/C++) can be combined in a single design, if needed. Because of this, one team member may prefer to work in VHDL while another uses Verilog. There may be project management reasons for discouraging multi-HDL use within a single project, but the languages themselves function together in a single design without issue. One reason a project manager may wish to avoid the use of multiple HDLs is that the future sustainment of the effort will require the participation of developers with skills in both languages.

Similarly, it is possible to define a high-level architecture for a project in terms of a block diagram and then implement detailed subsystem functionality using the HDL of choice. In the same design, it is also possible to integrate HDL generated from a C/C++ algorithm. Embedded system architects and developers should carefully consider the implications and select the appropriate implementation method for each portion of an FPGA design.

The next section will introduce the steps in the standard FPGA development process.

The FPGA development process

While FPGAs are used across a wide variety of disparate application domains, it is possible to identify a set of development steps that are broadly applicable to any FPGA development project. This section discusses the usual FPGA development steps in the sequence they normally occur during a project.

Defining system requirements

The first step in developing a new system, or when initiating a significant upgrade to an existing system, is to develop a clear and detailed understanding of what the system is supposed to do. The requirements definition process begins with a general description of the system's intended functionality, operating modes, and key features. This information should be written out in clear and unambiguous language and shared with all parties having a stake in the success of the development effort. The goal of sharing the system requirements is to achieve consensus among all of the parties as to the completeness and correctness of the descriptions.

Requirement descriptions must be fleshed out to include specifications for the required level of system performance in terms such as sampling rates of input signals and update rates for actuator output commands. Additional details such as physical size constraints, minimum battery lifetime, and tolerable environmental temperature ranges will guide the design process. In general, a comprehensive set of specifications must be developed that describes the minimum performance thresholds for all system parameters that are judged to be relevant to overall system success.

The full set of system requirements must be complete to the extent that any design solution that complies with all of the stated specifications must be an adequate solution. If it turns out that a design that satisfies all of the specifications is deemed unacceptable for some unrelated reason, this represents a failure to fully state the system requirements. For example, if a technically adequate solution is determined to be too expensive to produce, the source of the problem is likely to be a failure to fully define cost containment constraints during the requirements development process.

After the top-level system requirements have been defined and agreed upon, it is generally necessary to divide the overall system configuration into a collection of subsystems, each with a cohesive purpose and its own set of descriptive requirements and technical specifications. In a real-time embedded system architecture, the digital processing capability is likely to be represented as a subsystem with a corresponding collection of requirements

Allocating functionality to the FPGA

If the requirements for digital processing in a system architecture exceed the capabilities of microcontrollers and microprocessors that would otherwise be suitable for use in the system, it may be appropriate to consider incorporating an FPGA in the design. Some system architectures, particularly those that benefit from high-speed digital hardware performing parallel operations, are natural candidates for FPGA implementation. Other system architectures may be capable of adequate performance with traditional digital processing, but there may be valuable opportunities to take advantage of the flexibility and extensibility offered by an FPGA implementation over a planned lifetime that envisions substantial system upgrades in the future.

After the decision has been made to incorporate an FPGA in the design, the next step is to allocate the portions of overall system digital processing requirements to the FPGA device. This typically includes the specification of the FPGA input and output signals, the update rates of inputs and outputs, and the identification of components with which the FPGA must interact, including parts such as ADCs and RAM devices.

Identifying required FPGA features

Having defined the functions to be performed by the FPGA, and with knowledge of the interfaces to other devices that the FPGA must support, it becomes possible to develop a list of features that candidate FPGA devices must provide.

Some FPGA families are designed for low-cost, less-complex applications and thus offer a limited set of resources for implementing digital logic. These devices might operate from battery power and require only passive cooling. Other, more powerful, FPGA families support large-scale, full-featured digital designs, are intended to operate at peak performance, and may require continuous active cooling.

The system requirements associated with the embedded application will guide the selection of an appropriate FPGA family for the application. At this point, it is likely not possible to identify a specific FPGA model within the preferred family because the resource requirements of the FPGA implementation have not been fully defined. However, with experience, it is possible to identify a small number of FPGA models that appear suitable for the design.

In addition to the FPGA resources for digital circuit implementation, many FPGA models include additional features that may be important for the system design. For example, a built-in ADC may be useful for minimizing the system parts count. The list of required and desired FPGA features will help further narrow the selection of appropriate FPGA devices for the system.

Implementing the FPGA design

Having identified a candidate FPGA model, and with the detailed definition of the functionality allocated to the FPGA in hand, it is time to begin the implementation of the FPGA design. This will generally involve the use of the FPGA development tool suite and usually consists largely of developing HDL code in the preferred language for the project.

If appropriate, the FPGA implementation might begin with a block diagram representation of the top-level FPGA design. As necessary, components developed in HDL or C/C++ can be incorporated into the block design to complete the full system implementation.

Alternatively, it is also common for entire system designs to be developed directly in HDL. For developers familiar with the language and with a full understanding of the features and constraints of the FPGA model in use, this may lead to the most resource-efficient and highest-performing design outcome.

FPGA development proceeds in phases as the initial design becomes specified in more detail until a programming file for the FPGA device is produced. It is common to iterate through these phases several times for a large project, developing a small portion of the total design during each pass through the steps. These phases are described in the following sections.

Design entry

Design entry is the phase where the system developer defines system functionality using HDL code, block diagrams, and/or C/C++ code. The code and other artifacts, such as block diagrams, define the logical functionality of the system in abstract terms. In other words, the design artifacts define a logic circuit, but they don't define how it is integrated with the rest of the system.

I/O planning

FPGA I/O planning is the process of identifying the pins assigned to perform particular I/O functions and associating any device features such as the I/O signal standard to use for each signal. As part of the I/O planning process, it may be important to consider issues such as where on the physical device package I/O pins are located. This step is important to minimize the printed circuit board trace lengths for high-speed signals and to avoid forcing circuit signal traces to unnecessarily cross over one another.

The definition of I/O signal requirements is one form of **constraint** in the FPGA development process. The other primary constraint category consists of timing requirements that determine the FPGA solution's performance. The FPGA synthesis process uses the HDL code and the project constraints to develop a functionally correct FPGA solution that satisfies all of the defined constraints. If the tool cannot satisfy all of the constraints, synthesis will fail.

Synthesis

Synthesis transforms the source code into a circuit design called a **netlist**. The netlist represents the circuit constructed from the resources of the target FPGA model. The netlist represents a logical, or schematic, version of the circuit. It does not define how the circuit will be implemented in the physical FPGA device. This occurs in the next step.

Place and route

The **place** process takes the FPGA resources defined in the netlist and assigns them to specific logic elements within the selected FPGA. The resulting resource placements must satisfy any constraints that restrict the allocation of these elements, including I/O constraints and timing constraints.

After the logic elements have been assigned physical locations during the place process, a set of connections among the logic elements is configured during the **route** process. Routing implements all of the connections between the logic elements and enables the circuit to function as described in the HDL code. After the place and route operations have completed, the configuration of the FPGA is fully determined.

Bitstream generation

The final step in the FPGA development process is the production of a bitstream file. To achieve the highest performance, most modern FPGA devices store their configuration internally using **static RAM (SRAM)**.

You can think of the FPGA configuration SRAM as a very large shift register, containing perhaps millions of bits. The contents of this shift register fully specify all aspects of FPGA device configuration and operation. The bitstream file produced during FPGA development represents the settings for the shift register that cause the device to perform the intended functions specified by the HDL and the constraints. In terms of traditional software development processes, the bitstream file is analogous to an executable program produced by a linker.

SRAM is volatile and loses its contents each time device power is removed. The real-time embedded system architecture must provide a means for loading the bitstream file into the FPGA each time power is applied. Typically, the bitstream is either loaded from flash memory located within the device or from an external source, such as a PC, connected to the device during each power-on cycle.

Having completed the compilation of the FPGA bitstream, the next step is to test the implementation to verify that it operates correctly. This step is no different than the testing required at the end of a traditional software build process.

Testing the implementation

FPGA development is susceptible to all of the types of bugs that bedevil traditional software development efforts. During FPGA development, you will likely be presented with many error messages related to incorrect syntax, attempts to use resources not currently accessible, and many other types of violations. As in any programming endeavor, you will need to identify the source of each error and fix the problem.

Even after the FPGA application successfully proceeds through all of the stages to bitstream generation, there is no guarantee that the design will perform as intended. To achieve a successful design on a reasonable timetable, it is absolutely critical to perform adequate testing at each stage of development.

The first phase of testing should thoroughly exercise the behavior of the HDL code to demonstrate that it performs as intended. The example project at the end of this chapter will demonstrate the use of the Vivado tool suite to perform a thorough test of the HDL logic in the design.

After the bitstream has been generated, there is no substitute for comprehensive testing of the FPGA as implemented in the final system configuration. This testing must thoroughly exercise all features and modes of the FPGA, including its response to out-of-range and error conditions.

At each step of the design, development, and testing process, project personnel must remain attuned to the possibility of implementing system features that are susceptible to improper behavior in unlikely or rare situations. The occurrence of these kinds of issues can represent bugs that are extremely difficult to duplicate and that can forever tarnish the perception of the embedded system design and the organization that produced it. If you do an excellent job of testing, the likelihood of this outcome will be reduced substantially.

The next section provides a detailed description of the steps in the development, testing, and implementation of a simple FPGA project using the Arty A7 development board and the Xilinx Vivado tool suite.

In this section, we will develop and implement a simple but complete project using a Xilinx Artix-7 FPGA device installed on a Digilent Arty A7 development board. This board comes in two variants, a lower-cost version (US \$129) with a model number ending in -35T and a more capable, but more costly, version (US \$249) with a model number ending in -100T. The only difference between the two boards is the model of the Artix-7 FPGA installed on the board. As you would expect, the -35T has fewer resources available than the -100T.

You can use either the -35T or the -100T variant for this project. The only difference in the development process is specifying the correct board model whenever the need arises. However, in later chapters, the -100T variant will be required due to the resource requirements of the example digital oscilloscope project design, so the more capable board is recommended.

The Arty A7 boards are available for purchase at <https://store.digilentinc.com/artix-a7-artix-7-fpga-development-board-for-makers-and-hobbyists/> and from other sources, such as Amazon.

For the purpose of this project, the resources on the board of interest are the FPGA device itself, as well as the four switches, four pushbuttons, and five LEDs. This project will demonstrate how to install the Vivado tool suite, create a project, enter HDL code, test the code, and ultimately produce a bitstream and download it to the board. After downloading the bitstream to the board, you will be able to manually test the operation of the system. You will also see how to program the FPGA image into flash memory on the Arty A7 board so that it loads and runs each time the board powers on.

Project description

This project will implement a four-bit binary adder in the FPGA. This is intentionally a very simple design because the focus here is on setting up the tools and learning how to use them, and not on implementing a complex HDL model.

The four switches on the board represent one 4-bit binary number and the four pushbuttons represent another 4-bit number. The FPGA logic will continuously perform an addition operation between these two numbers and display the result as a 4-bit binary number on four LEDs with a fifth LED representing the carry bit.

The 4-bit adder code is based on the single-bit full adder circuit described in the *Hardware design languages* section of *Chapter 1, Architecting High-Performance Embedded Systems*.

Installing the Vivado tools

We will use the Xilinx Vivado suite of FPGA development tools for this project and for projects in future chapters. These tools are available for free and are supported on Windows and Linux operating systems. You may install the tools on either operating system. The description in this section covers the Windows version of the tools, but if you are installing on Linux, the differences should be obvious. Working with the Vivado tools should be nearly identical on the different operating systems:

1. If you don't already have one, create a Xilinx user account at <https://www.xilinx.com/registration/create-account.html>.
2. Visit <https://xilinx.com> and log in to your user account. Once logged in, go to the tools download page at <https://www.xilinx.com/support/download.html>.
3. Download the **Xilinx Unified Installer: Windows Self-Extracting Web Installer**. You should probably select the latest version available, but if you want to follow along with the version used in this book, select version **2020.1**.
4. The installer file will have a name similar to `Xilinx_Unified_2020.1_0602_1208_Win64.exe`. Locate this file in your downloads directory and run it. If a dialog warns you about installing an app that isn't Microsoft-verified, click **Install anyway**.
5. When the **Welcome** screen comes up, click **Next**:

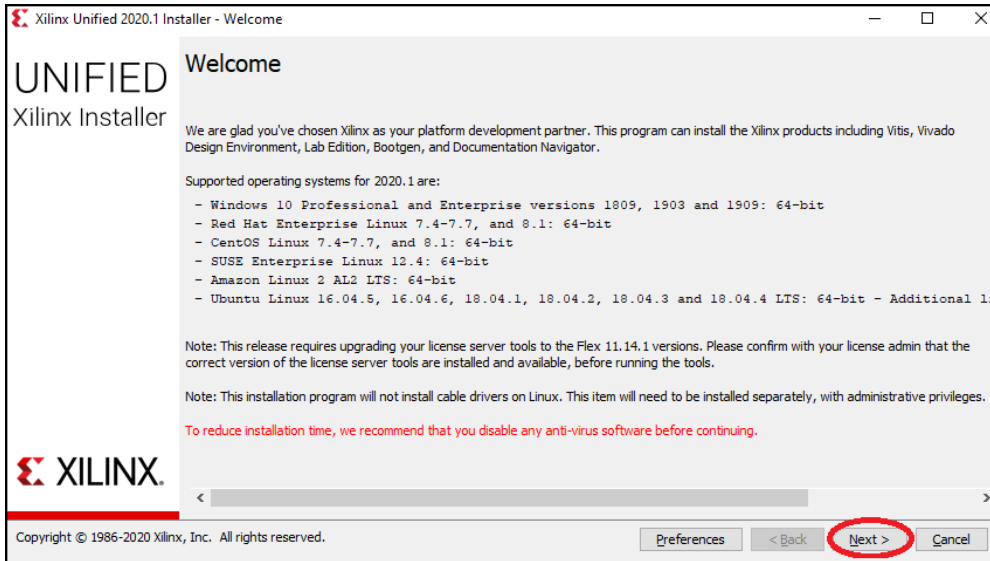


Figure 4.3 – Installer Welcome dialog

6. On the following screen, enter your `xilinx.com` user ID and password, then click **Next**:

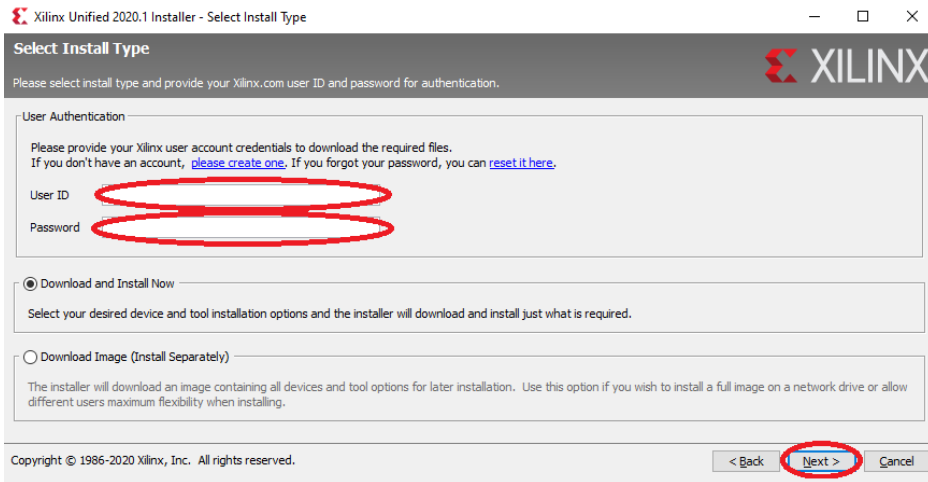


Figure 4.4 – Installer login dialog

7. The next dialog requests that you accept some license agreements. Check the boxes that say **I Agree**, then click **Next**.
8. In the next dialog, leave **Vitis** selected as the product to be installed and click **Next**. Vitis includes the Vivado tool suite along with a collection of other Xilinx development tools:

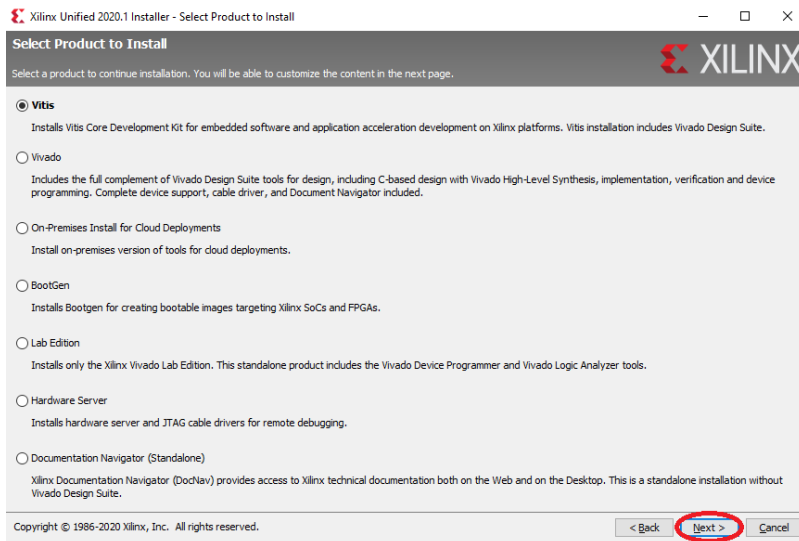


Figure 4.5 – Installer product selection dialog

9. The next dialog allows you to select the software components to be installed. Leave the selections at their default values and click **Next**.
10. The next dialog allows you to select a destination directory and specify program shortcut options. A destination directory of `C:\Xilinx` is a suitable location. Create this directory if it does not exist. Click **Next**.
11. The next dialog displays a summary of the installation options. Click **Install** to proceed with the installation. Depending on the speed of your computer and your internet connection, installation may take a few hours to complete:

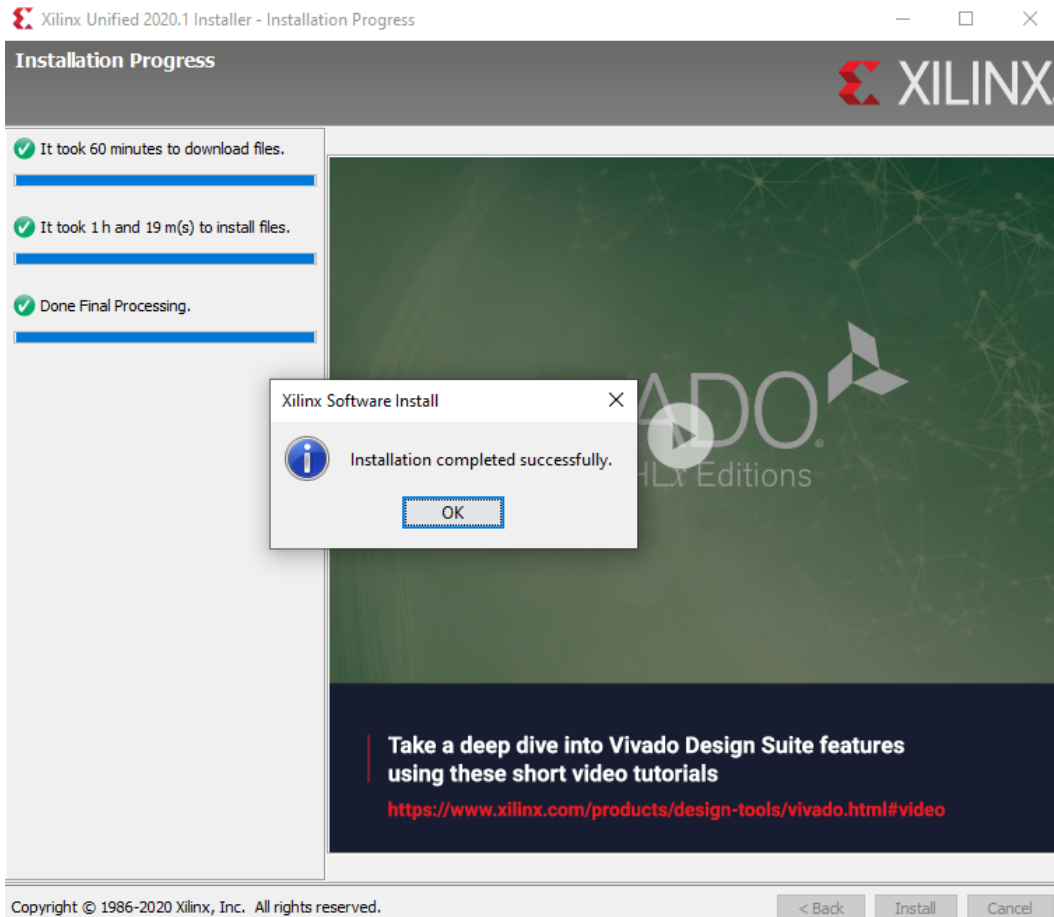


Figure 4.6 – Installation complete dialog

Having completed the installation, we will next create our first project.

Creating a project

Follow these steps to create and build the 4-bit binary adder project for the Arty A7 board:

1. Locate the desktop icon titled **Vivado 2020.1** (or look for your version number, if different) and double-click it.
2. When Vivado displays its main screen, click **Create Project** in the **Quick Start** section:

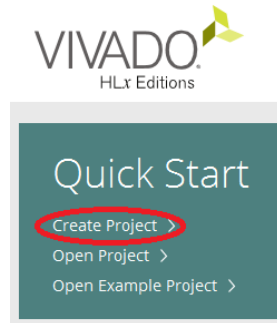


Figure 4.7 – Vivado Quick Start dialog

3. This will start the **Create a New Vivado Project** wizard. Click **Next** to reach the **Project Name** page and enter `ArtyAdder` as the project name. Select an appropriate directory location for the project and check the box to create a subdirectory, then click **Next**. Examples in this book will use the `C:\Projects` directory as the location for all projects:

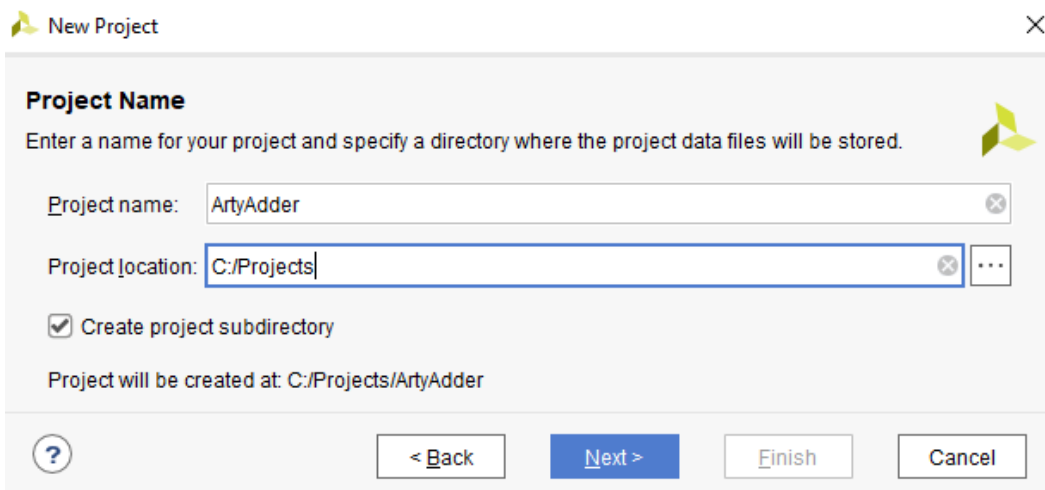


Figure 4.8 – Project Name dialog

- In the **Project Type** dialog, select **RTL Project** and check the box next to **Do not specify sources at this time**. Click **Next**:

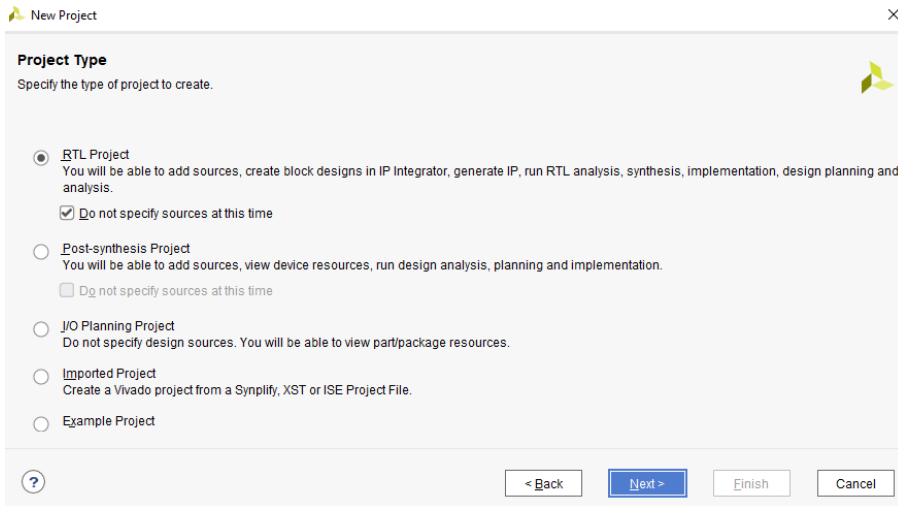


Figure 4.9 – Project Type dialog

- In the **Default Part** dialog, click the **Boards** tab and type **Arty** into the **Search** field. Depending on the board type you have (or if you don't have a board yet), select either the **Arty A7-100** or **Arty A7-35** and click **Next**:

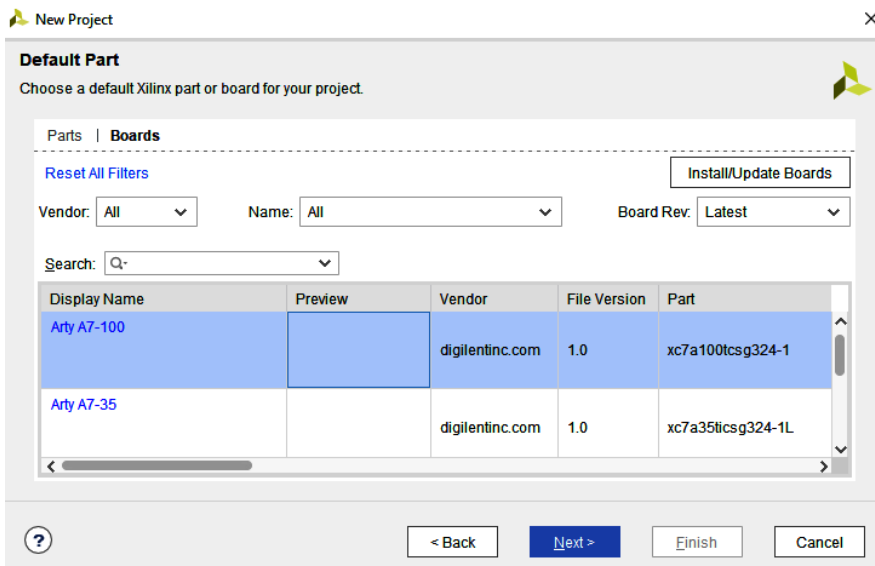


Figure 4.10 – Default Part dialog

- In the **New Project Summary** dialog, click **Finish**.

We have now created an empty project. In the next section, we will create VHDL source files containing the logic circuit design for this project.

Creating VHDL source files

The following steps describe the process of creating VHDL source files, entering source code, and compiling the FPGA design:

1. In the **Sources** sub-window, right-click **Design Sources** and select **Add Sources...**:

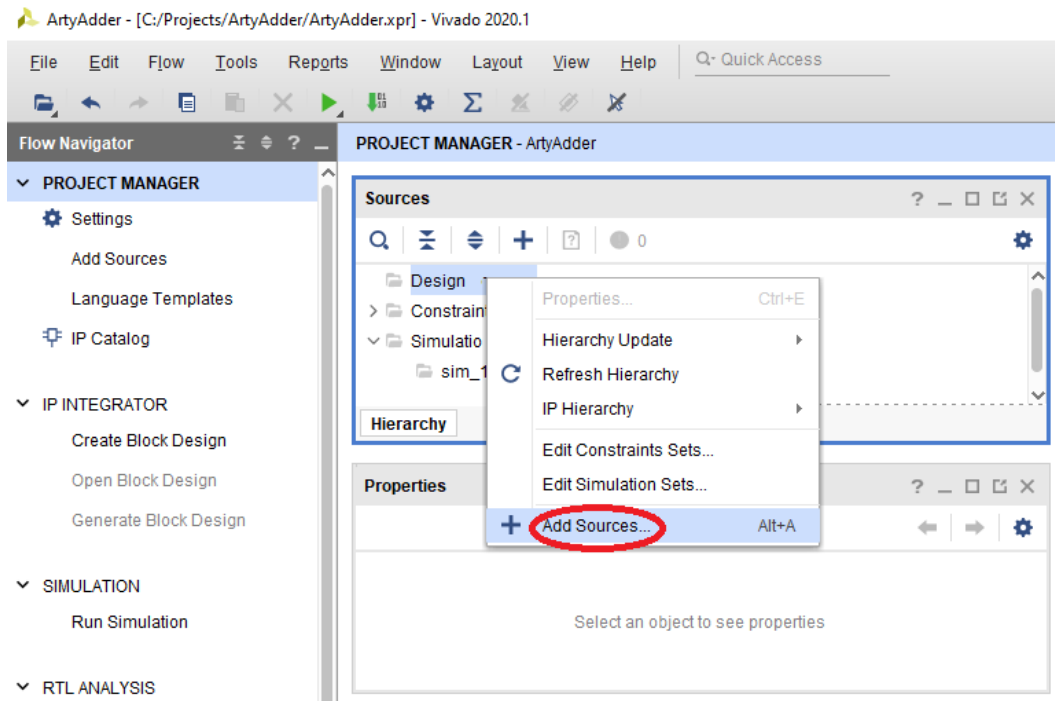


Figure 4.11 – Add Sources... menu selection

2. In the **Add Sources** dialog, ensure **Add or create design sources** is selected, then click **Next**.
3. In the **Add or Create Design Sources** dialog, click **Create File**:

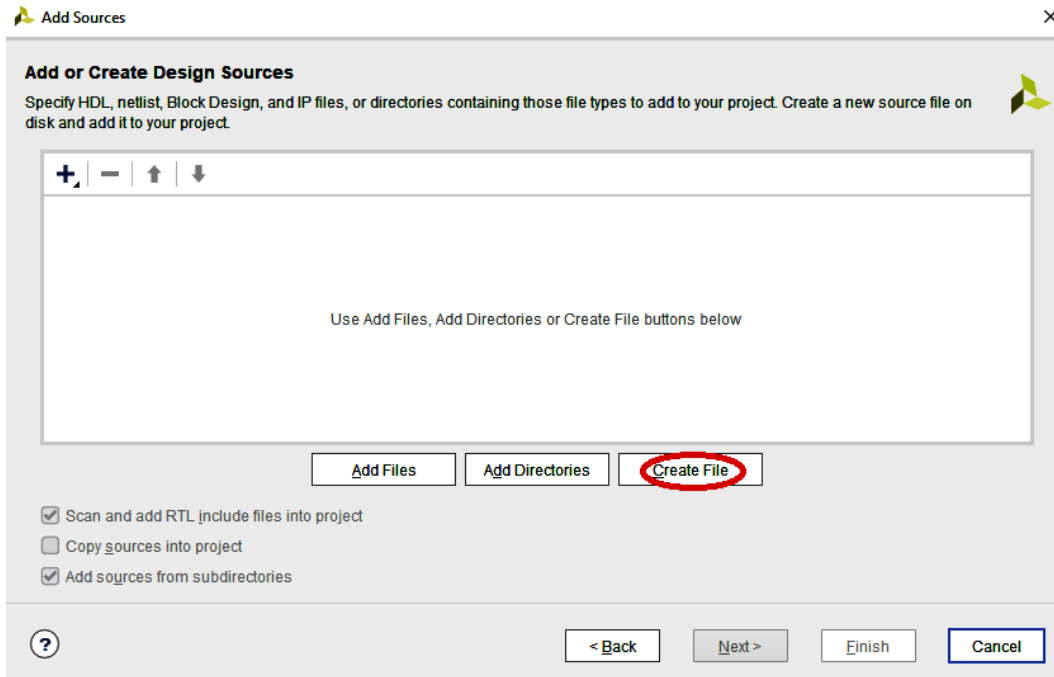


Figure 4.12 – Add or Create Design Sources dialog

4. Enter the filename `FullAdder.vhdl` and click **OK**:

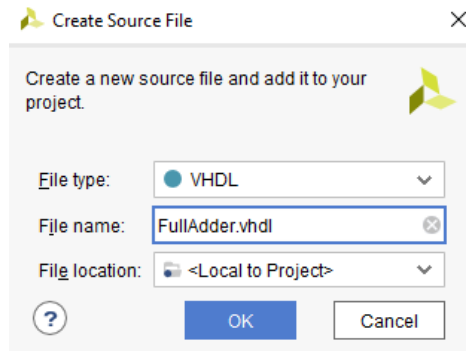


Figure 4.13 – Create Source File dialog

5. Repeat the previous two steps to create another file named `Adder4.vhdl`, then click **Finish** in the **Add or Create Design Sources** dialog.

6. The **Define Modules** dialog will appear next. We will not be entering anything here. Click **OK** to close this dialog. You will be asked if you are sure you want to use these values. Click **Yes**:

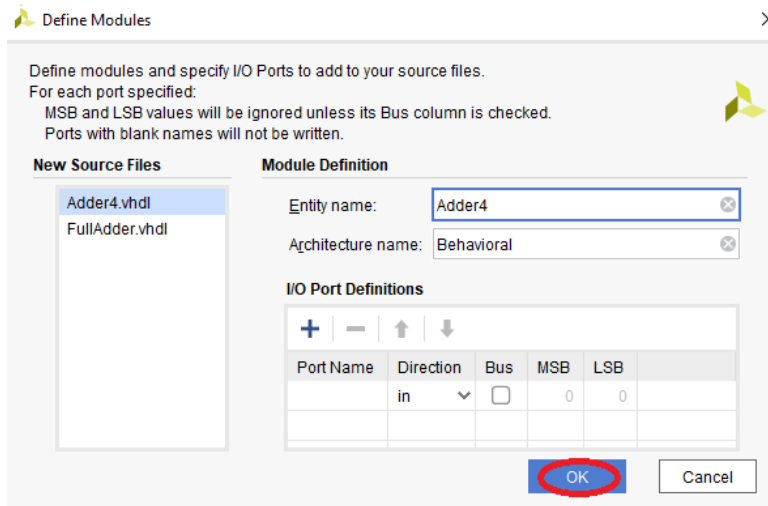


Figure 4.14 – Define Modules dialog

7. Expand the **Non-module Files** under **Design Sources**, then double-click `FullAdder.vhdl`. An editor window will open displaying the empty `FullAdder.vhdl` file:

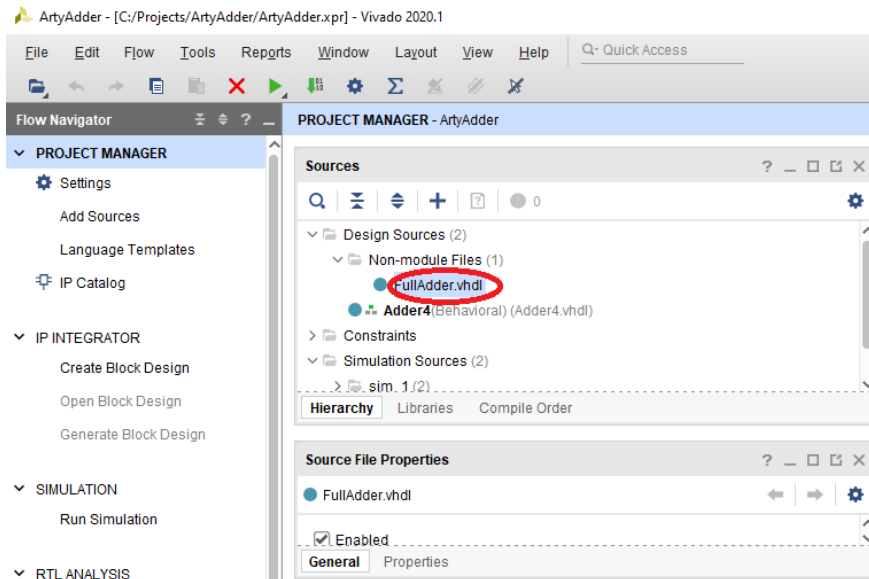


Figure 4.15 – Newly created source file

8. Enter the following VHDL code into the `FullAdder.vhdl` editor window:

```
-- Load the standard libraries

library IEEE;
    use IEEE.STD_LOGIC_1164.ALL;

-- Define the full adder inputs and outputs

entity FULL_ADDER is
    port (
        A      : in    std_logic;
        B      : in    std_logic;
        C_IN   : in    std_logic;
        S      : out   std_logic;
        C_OUT  : out   std_logic
    );
end entity FULL_ADDER;

-- Define the behavior of the full adder

architecture BEHAVIORAL of FULL_ADDER is

begin

    S      <= (A XOR B) XOR C_IN;
    C_OUT <= (A AND B) OR ((A XOR B) AND C_IN);

end architecture BEHAVIORAL;
```

This is the same single-bit full adder code we examined in the *Hardware design languages* section of *Chapter 1, Architecting High-Performance Embedded Systems*. Figure 4.16 shows the code in the Vivado editor window:

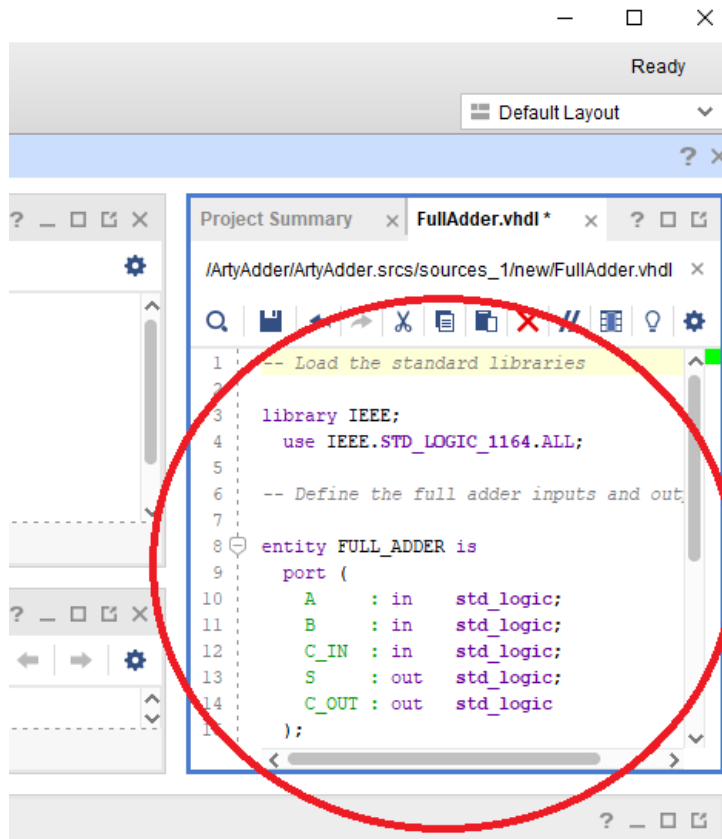


Figure 4.16 – FullAdder.vhdl source code

9. In the same manner, double-click **Adder4(Behavioral)** (**Adder4.vhdl**) under **Design Sources**. Delete the automatically populated contents of the **Adder4.vhdl** editor window and enter the following code into the **Adder4.vhdl** editor:

```
-- Load the standard libraries

library IEEE;
    use IEEE.STD_LOGIC_1164.ALL;

-- Define the 4-bit adder inputs and outputs
```

```

entity ADDER4 is
  port (
    A4      : in    std_logic_vector(3 downto 0);
    B4      : in    std_logic_vector(3 downto 0);
    SUM4     : out   std_logic_vector(3 downto 0);
    C_OUT4   : out   std_logic
  );
end entity ADDER4;

-- Define the behavior of the 4-bit adder

architecture BEHAVIORAL of ADDER4 is

  -- Reference the previous definition of the full adder

  component FULL_ADDER is
    port (
      A          : in    std_logic;
      B          : in    std_logic;
      C_IN       : in    std_logic;
      S          : out   std_logic;
      C_OUT      : out   std_logic
    );
  end component;

  -- Define the signals used internally in the 4-bit adder
  signal c0, c1, c2 : std_logic;

begin

  -- The carry input to the first adder is set to 0
  FULL_ADDER0 : FULL_ADDER
    port map (
      A      => A4(0),
      B      => B4(0),
      C_IN   => '0',

```



```
        S          => SUM4 (0) ,
        C_OUT      => c0
    );

    FULL_ADDDER1 : FULL_ADDDER
    port map (
        A          => A4 (1) ,
        B          => B4 (1) ,
        C_IN       => c0 ,
        S          => SUM4 (1) ,
        C_OUT      => c1
    );

    FULL_ADDDER2 : FULL_ADDDER
    port map (
        A          => A4 (2) ,
        B          => B4 (2) ,
        C_IN       => c1 ,
        S          => SUM4 (2) ,
        C_OUT      => c2
    );

    FULL_ADDDER3 : FULL_ADDDER
    port map (
        A          => A4 (3) ,
        B          => B4 (3) ,
        C_IN       => c2 ,
        S          => SUM4 (3) ,
        C_OUT      => C_OUT4
    );

end architecture BEHAVIORAL;
```

This code instantiates four copies of the single-bit full adder. The carry into the least significant adder is set to zero and the carry from each adder ripples to the next most-significant adder. The result of adding two 4-bit numbers is a 4-bit result and a single-bit carry:

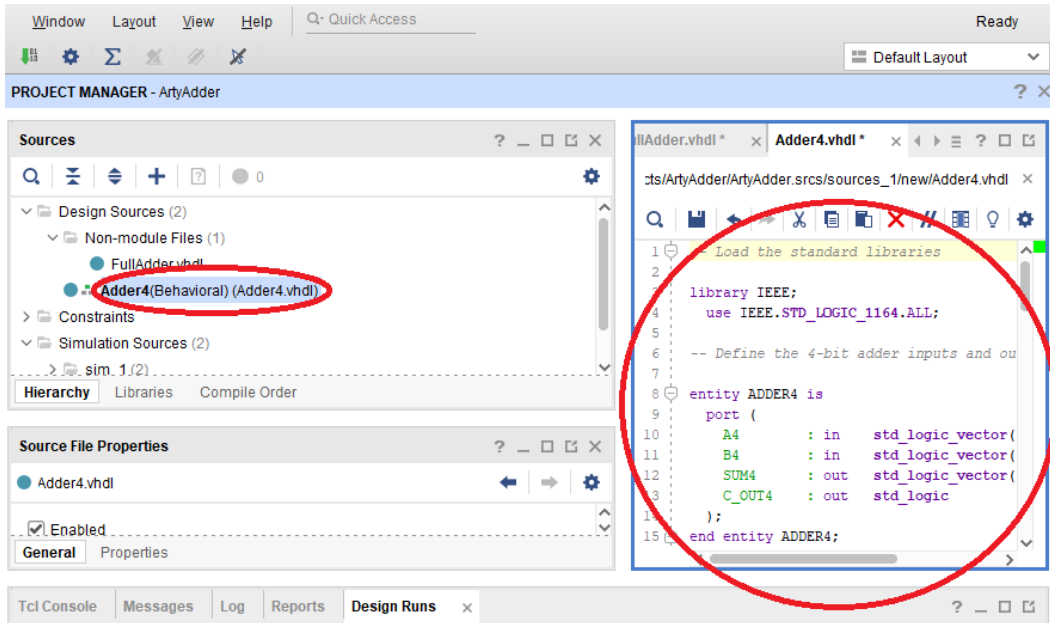


Figure 4.17 – Adder4.vhdl source code

At this point, you have entered VHDL code that defines a 4-bit binary adder constructed from four single-bit full adders. Next, we will test the correctness of the implementation.

Testing the logic behavior

It is important to test the behavior of logic using simulation before trying to run it in the FPGA. This is because it is much easier to detect and fix problems in the simulation environment than it is with the logic running inside the FPGA. The Vivado simulation tools do a very good job of representing circuit behavior:

1. In the **Sources** sub-window, right-click on **Simulation Sources** and select **Add Sources...**:

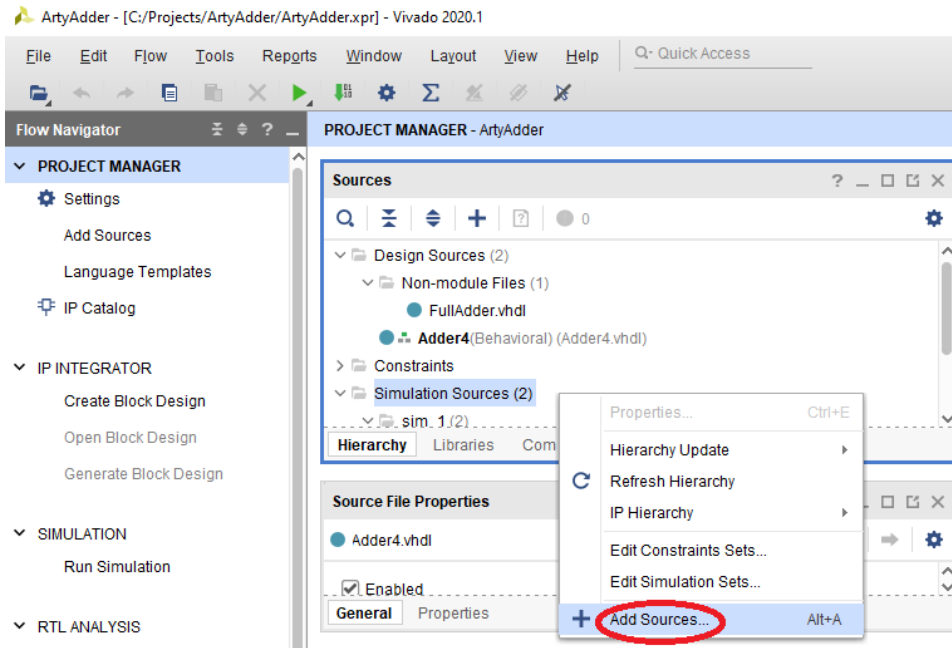


Figure 4.18 – Add Sources... menu selection for Simulation Sources

2. In the **Add Sources** dialog, ensure **Add or create simulation sources** is selected, then click **Next**.
3. In the **Add or Create Simulation Sources** dialog, click **Create File**.
4. Enter the filename `Adder4TestBench.vhdl` and click **OK**.
5. Click **Finish** to dismiss the **Add or Create Simulation Sources** dialog, then click **OK** in the **Define Module** dialog and click **Yes** when asked if you are sure you want to use these values.

6. Double-click **Adder4 TestBench (Behavioral) (Adder4TestBench.vhdl)** under **Simulation Sources**. Delete the automatically populated contents of the **Adder4TestBench.vhdl** editor window and enter the following code into the **Adder4TestBench.vhdl** editor:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity ADDER4_TESTBENCH is
end entity ADDER4_TESTBENCH;

architecture BEHAVIORAL of ADDER4_TESTBENCH is

    component ADDER4 is
        port (
            A4      : in    std_logic_vector(3 downto 0);
            B4      : in    std_logic_vector(3 downto 0);
            SUM4     : out   std_logic_vector(3 downto 0);
            C_OUT4   : out   std_logic
        );
    end component;

    signal a          : std_logic_vector(3 downto 0);
    signal b          : std_logic_vector(3 downto 0);
    signal s          : std_logic_vector(3 downto 0);
    signal c_out      : std_logic;

    signal expected_sum5 : unsigned(4 downto 0);
    signal expected_sum4 : unsigned(3 downto 0);
    signal expected_c    : std_logic;
    signal error         : std_logic;

begin

    TESTED_DEVICE : ADDER4
        port map (
```

```
A4      => a,
B4      => b,
SUM4    => s,
C_OUT4  => c_out
);

TEST : process
begin

    -- Test all combinations of two 4-bit addends (256 total
    tests)
    for a_val in 0 to 15 loop
        for b_val in 0 to 15 loop
            -- Set the inputs to the ADDER4 component
            a <= std_logic_vector(to_unsigned(a_val, a'length));
            b <= std_logic_vector(to_unsigned(b_val, b'length));
            wait for 1 ns;

            -- Compute the 5-bit sum of the two 4-bit values
            expected_sum5 <= unsigned('0' & a) + unsigned('0' & b);
            wait for 1 ns;

            -- Break the sum into a 4-bit output and a carry bit
            expected_sum4 <= expected_sum5(3 downto 0);
            expected_c     <= expected_sum5(4);
            wait for 1 ns;

            -- The 'error' signal will only go to 1 if an error
            occurs
            if ((unsigned(s) = unsigned(expected_sum4)) and
                (c_out = expected_c)) then
                error <= '0';
            else
                error <= '1';
            end if;

            -- Each pass through the inner loop takes 10 ns
```

```
        wait for 7 ns;  
  
    end loop;  
end loop;  
  
wait;  
  
end process TEST;  
  
end architecture BEHAVIORAL;
```

This code exercises the 4-bit adder functionality by presenting all combinations of 4-bit numbers to each of the A4 and B4 inputs to the Adder4 component. It compares the SUM4 and C_OUT4 outputs of the Adder4 component to independently computed values for the same inputs. After each addition operation, the error signal is set to 0 if the Adder4 outputs matched the expected values, or it is set to 1 if there is a mismatch.

The code in `Adder4TestBench.vhdl` resembles traditional software code in the way it uses nested `for` loops to apply all of the test input combinations to the Adder4 component under test. Code that runs tests in simulation mode is non-synthesizable, which means it does not purely represent a hardware logic circuit and is capable of traditional software-like operations, such as the iterative execution of `for` loops.

However, as in physical circuits, signals being assigned values in the test bench code using the `<=` operator cannot be used at the same instant in time in subsequent expressions. This is because the simulation environment represents the real-world effects of propagation delay, which is significant even within tiny FPGA devices. The three `wait for 1 ns;` statements in the test bench code pause circuit operations to allow for propagation delay. These 1 ns delays provide time for the signal values computed just before the `wait` statement to propagate so they can be used in the following statement. The final `wait for 7 ns;` statement in the inner loop is a pause that allows us to clearly see the results of each iteration of the simulation loops in the signal trace display.

7. Right-click **Adder4 TestBench (Behavioral)** (**Adder4TestBench.vhdl**) under **Simulation Sources** and select **Automatic Update and Compile Order**. This sets **ADDER4_TESTBENCH** as the top-level object for the simulation run:

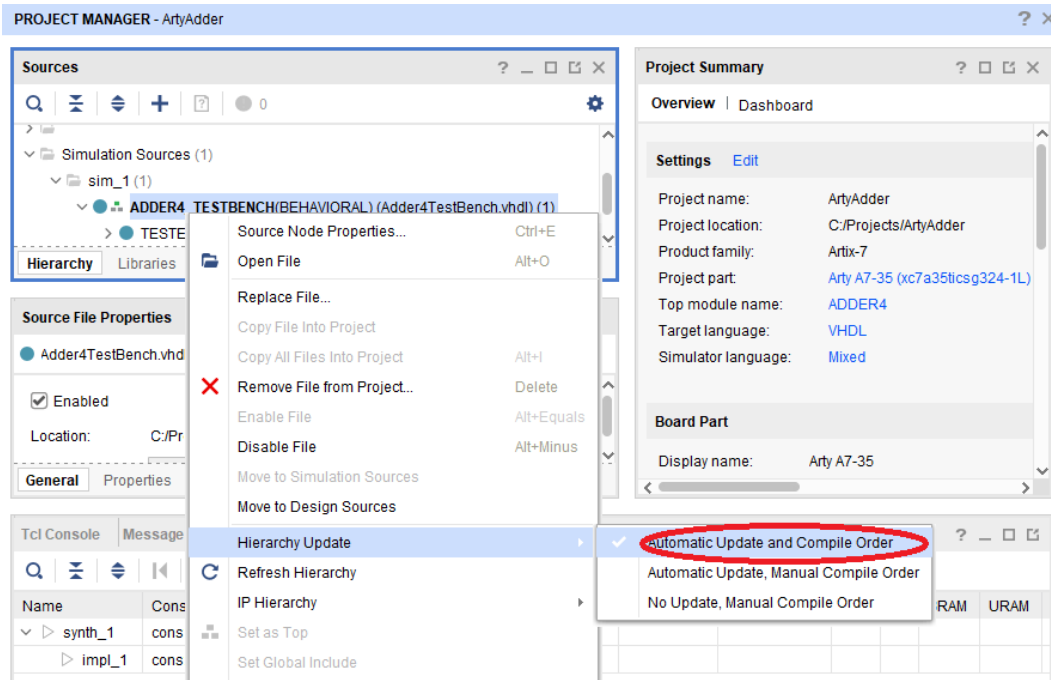


Figure 4.19 – Menu selection to set Automatic Update and Compile Order

8. Click **Run Simulation**, then **Run Behavioral Simulation** in the **Flow Navigator** window to enter simulation mode. If you haven't already saved the editor files, you will be prompted to do so. Click **Save**. The simulation will then run:

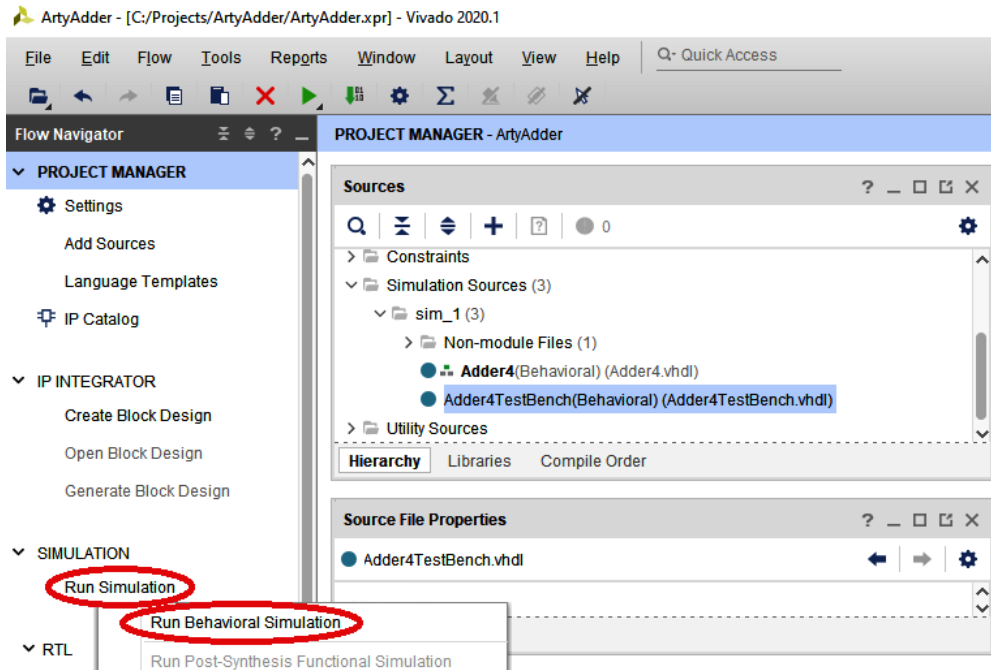


Figure 4.20 – Run Behavioral Simulation menu selection

9. When the **SIMULATION** window opens, click the maximize button in the simulation output window with the title **Untitled 1**:

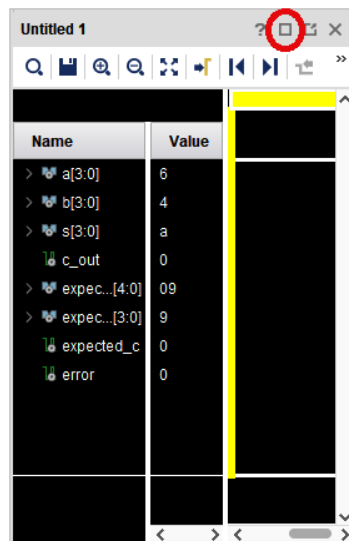


Figure 4.21 – Simulation results window

The total simulated time of each pass through the inner loop is 10 ns. Because there are 256 passes through the loop in `Adder4TestBench.vhdl`, the time to run the simulation is 2560 ns.

10. Set the simulation run time to 2560 ns in the top toolbar (*step 1* in the following figure), press the left-pointing restart button (*step 2*), then press the right-facing button to run the simulation for 2560 ns (*step 3*), and, finally, press the *Zoom Fit* button (*step 4*) to scale the simulation output data range to fit the window:

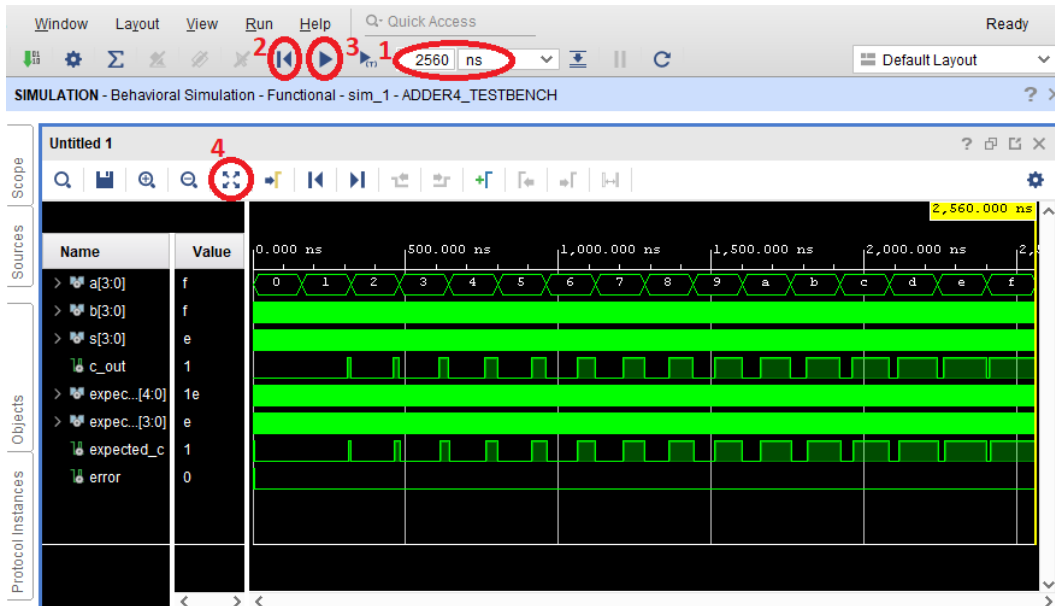


Figure 4.22 – Simulation results from the start to the end of the run

You can use the magnifier icons to zoom in on any point of the trace and observe the results of each addition operation performed during testing. For example, the following figure shows the decimal values 6 and 2 were added to produce the result 8 with a carry of 0. These values match the expected values, which caused error to be set to 0. The error signal is 0 for all 256 test cases, indicating our logic circuit passed all of the tests:

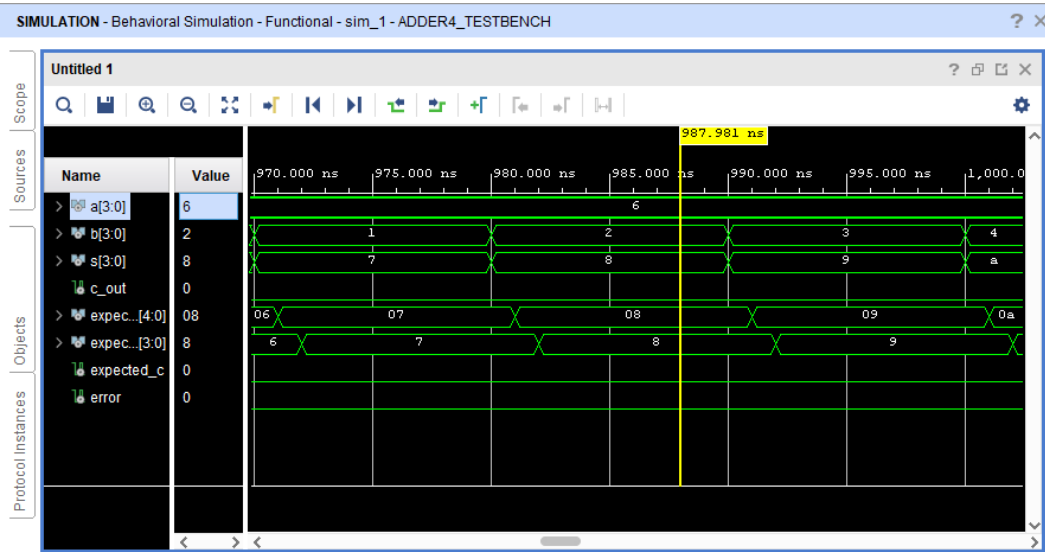


Figure 4.23 – Zoomed-in view of the simulation results

11. Close simulation mode by clicking the **X** in the blue **SIMULATION** bar above the data trace window. Click **OK** when asked if you want to close the simulation.

Having passed behavioral testing, we will define the I/O signals used in the design.

Defining I/O signals

Our next step is to connect the inputs and outputs of our circuit to hardware devices on the Arty board. The inputs will be the board switches and pushbuttons, and the outputs will be LEDs.

The following steps will create a constraints file that describes the I/O pins we will use on the FPGA device and the functions connected to those pins on the Arty board. Constraint files have the `.xdc` extension:

1. In the **Sources** sub-window, right-click **Constraints** and select **Add Sources...**
2. In the **Add Sources** dialog, ensure **Add or create constraints** is selected, then click **Next**.
3. In the **Add or Create Constraints** dialog, click **Create File**.
4. Enter the filename `Arty-A7-100.xdc` (or `Arty-A7-35.xdc` if appropriate for your device) and click **OK**.
5. Click **Finish** to dismiss the **Add or Create Constraints** dialog.

6. Expand the **Constraints** source tree and double-click **Arty-A7-35.xdc**.
7. Digilent provides pre-populated constraint files for the Arty A7 boards online. Visit <https://raw.githubusercontent.com/Digilent/digilent-xdc/master/Arty-A7-35-Master.xdc> and copy the entire content of the browser window into the Arty-A7-35.xdc editor window in Vivado. If appropriate for your device, use the file at <https://raw.githubusercontent.com/Digilent/digilent-xdc/master/Arty-A7-100-Master.xdc> instead.
8. All of the I/O pins are commented out in the constraints file by default. Uncomment the appropriate lines in the file by removing the # character from the beginning of each line. We will be using the pins listed in the following sections in the Arty-A7-100.xdc file: Switches, RGB LEDs (but only led0_g, the first green LED), LEDs, and Buttons. The following figure shows these lines after they have been uncommented:

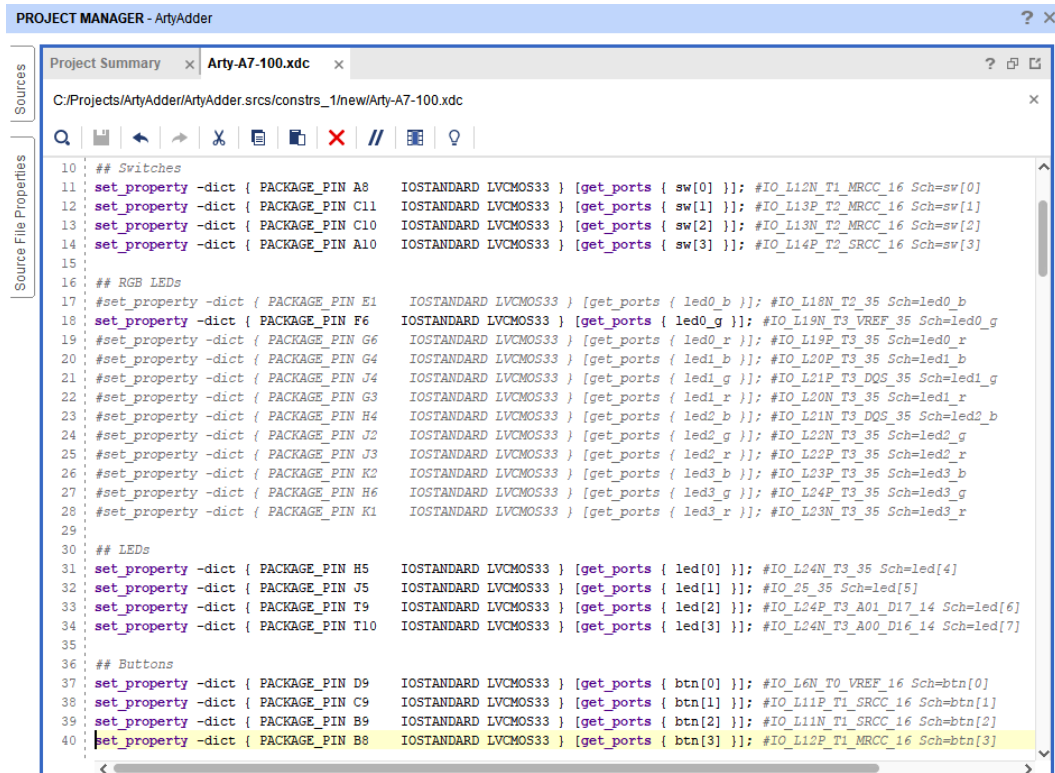


Figure 4.24 – Constraints editor window

In the next section, we will create a top-level VHDL file that interfaces the adder code with the I/O devices.

Creating a top-level VHDL file

We will next create a top-level VHDL file that connects our 4-bit adder component to the corresponding board I/O signals:

1. In the **Sources** sub-window, right-click on **Design Sources** and select **Add Sources....**
2. In the **Add Sources** dialog, ensure **Add or create design sources** is selected, then click **Next**.
3. In the **Add or Create Design Sources** dialog, click **Create File**.
4. Enter the filename `ArtyAdder.vhdl` and click **OK**.
5. Click **Finish** to dismiss the **Add or Create Design Sources** dialog, then click **OK** in the **Define Module** dialog and click **Yes** when asked if you are sure you want to use these values.
6. Double-click **ArtyAdder.vhdl** under **Design Sources**. Delete the automatically populated contents of the **ArtyAdder.vhdl** editor window and enter the following code into the **ArtyAdder.vhdl** editor:

```
-- Load the standard libraries

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity ARTY_ADDER is
    port (
        sw          : in  STD_LOGIC_VECTOR (3 downto 0);
        btn         : in  STD_LOGIC_VECTOR (3 downto 0);
        led         : out STD_LOGIC_VECTOR (3 downto 0);
        led0_g      : out STD_LOGIC
    );
end entity ARTY_ADDER;

architecture BEHAVIORAL of ARTY_ADDER is

    -- Reference the previous definition of the 4-bit adder

    component ADDER4 is
```

```
port (  
    A4      : in    std_logic_vector(3 downto 0);  
    B4      : in    std_logic_vector(3 downto 0);  
    SUM4    : out   std_logic_vector(3 downto 0);  
    C_OUT4  : out   std_logic  
);  
  
end component;  
  
  
begin  
    ADDER : ADDER4  
        port map (  
            A4      => sw,  
            B4      => btn,  
            SUM4    => led,  
            C_OUT4  => led0_g  
        );  
  
  
end architecture BEHAVIORAL;
```

This code maps the signal names for the I/O devices named in `Arty-A7-100.xdc` as `sw` (4 switches), `btn` (4 pushbuttons), `led` (4 single-color LEDs), and `led0_g` (the green channel of the first multi-color LED) to the `ADDER4` inputs and outputs.

While VHDL is not case-sensitive, the processing of `xdc` constraint files in Vivado is case-sensitive. The case used in I/O device names defined in the `xdc` file must be identical when referenced in a VHDL file. Specifically, the I/O signal names in VHDL must be lowercase in this file because they are lowercase in the constraints file.

We are now ready to synthesize, implement, and program our design for the Arty board.

Synthesizing and implementing the FPGA bitstream

If you wish, you can separately perform the synthesis and the implementation (place and route) steps using the selections in the **Flow Navigator** portion of the Vivado main dialog.

Alternatively, you can select **Generate Bitstream** and Vivado will perform all of the required steps, including synthesis, implementation, and bitstream generation without further user intervention. If a fatal error occurs, the process will stop and error messages will be displayed. Perform the following steps to generate the bitstream:

1. Click **Generate Bitstream** to start the build process. You may be asked if you want to save text editors. Click **Save**. You may be informed that there are no implementation results available and asked if it is OK to launch synthesis and implementation. Click **Yes**:

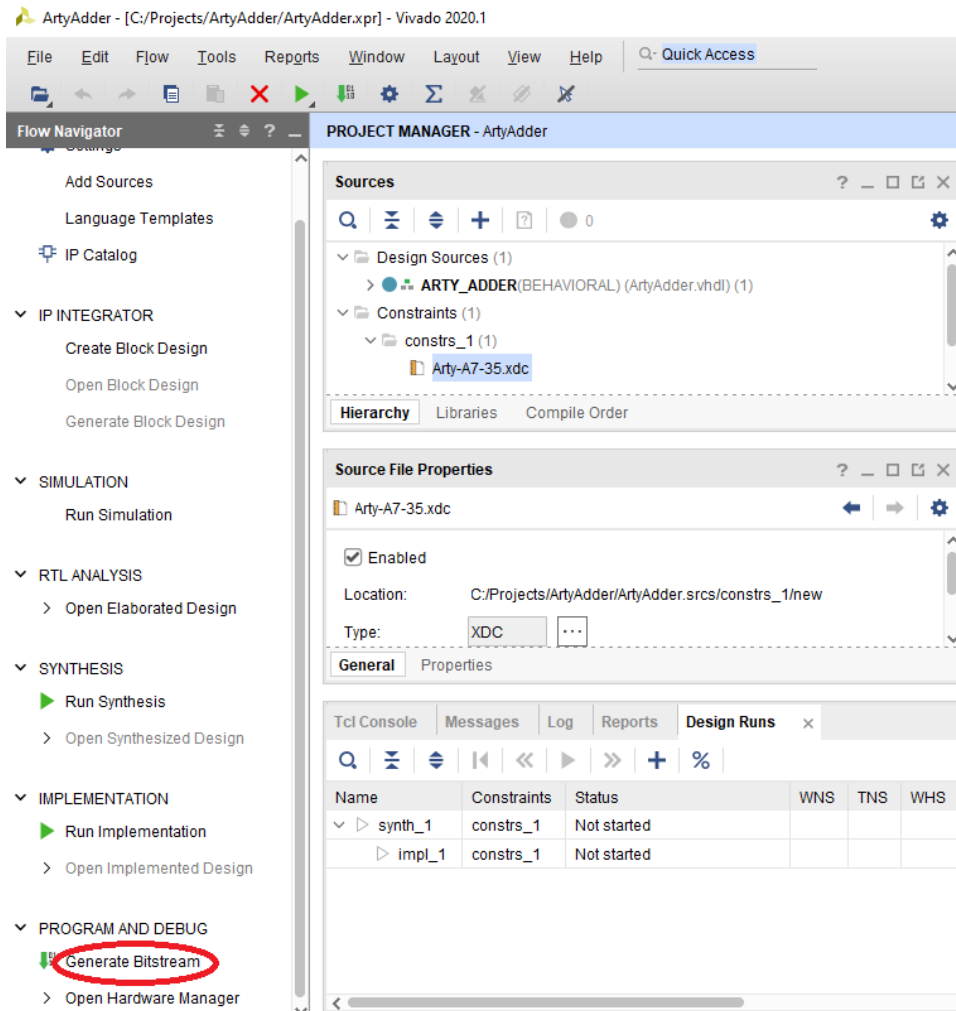


Figure 4.25 – Generate Bitstream menu selection

- The **Launch Runs** dialog will then appear. You can select a value for **Number of jobs** up to the number of processor cores in your computer. Using more cores makes the process go faster, but it can bog down your machine if you want to continue using it during a lengthy build process. Click **OK** to start the build:

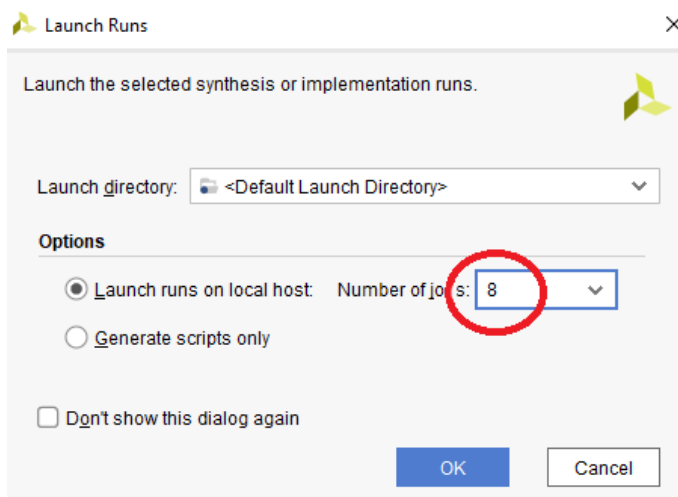


Figure 4.26 – Launch Runs dialog

- During the build process, Vivado will display the status in the upper-right corner of the main window. If necessary, you can cancel the build process by clicking **Cancel** next to the status display:



Figure 4.27 – Compilation status display

- When the build process completes, assuming there were no fatal errors, a **Bitstream Generation Completed** dialog will appear. Although other options are offered, we will proceed directly to downloading the bitstream to the Arty board. Select **Open Hardware Manager** and click **OK**:

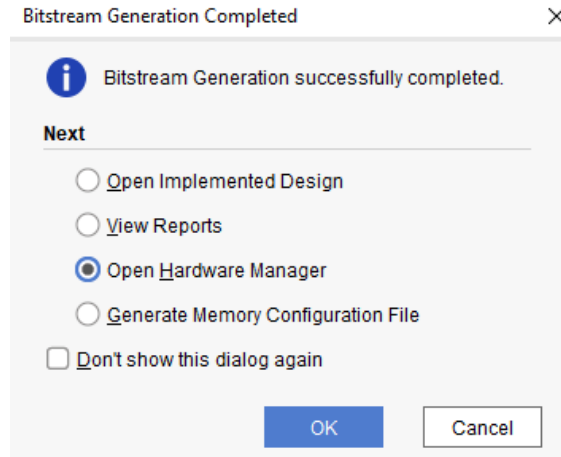


Figure 4.28 – Bitstream Generation Completed dialog

Next, we will download the bitstream into the FPGA.

Downloading the bitstream to the board

Perform the following steps to download the bitstream to an Arty A7 board:

1. The **HARDWARE MANAGER** dialog will appear and indicate **No hardware target is open**.
2. Connect your Arty A7-35 or A7-100 board to the computer with a USB cable. Wait a few seconds for the board to be recognized, then click **Open target**, then **Auto Connect**:

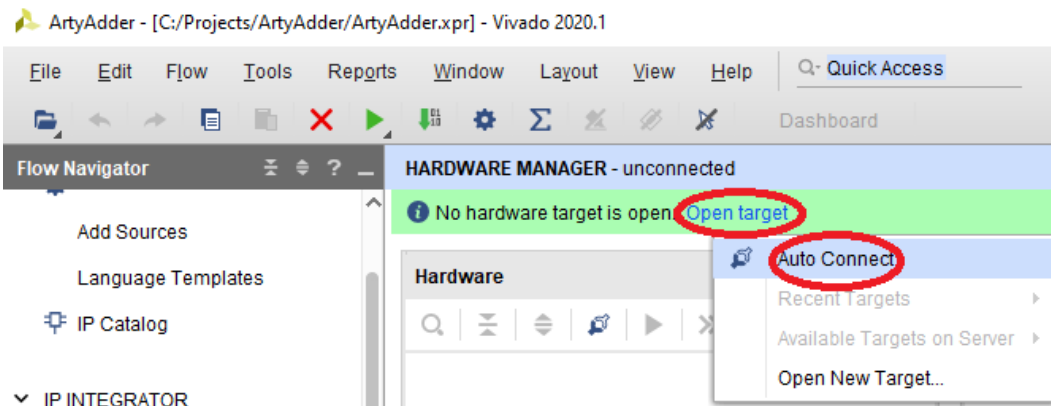


Figure 4.29 – Open target and Auto Connect selections

3. After a few seconds, Vivado should indicate that the board is connected. Click **Program device** to download the FPGA bitstream to the Arty board. You will be prompted to select a bitstream file. If you've used the same directory structure as this example, the file will be located at `C:/Projects/ArtyAdder/ArtyAdder.runs/impl_1/ARTY_ADDER.bit`:

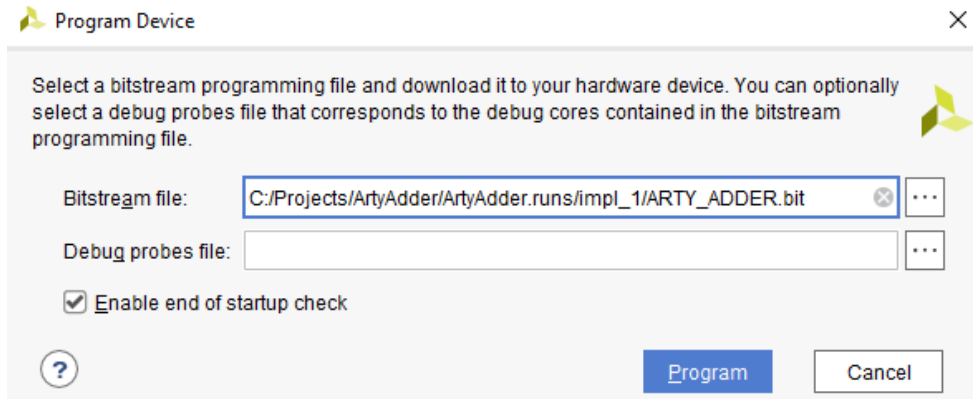


Figure 4.30 – Program Device dialog

4. Click **Program** to download the program to the FPGA device and start it executing.
5. You can now test the operation of the program with the Arty I/O devices. Place all of the four switches in the *off* position (move the switches toward the adjacent board edge) and do not press any of the four pushbuttons. All of the four green LEDs should be off.
6. If you turn on any individual switch or press any one pushbutton, the corresponding green LED should turn on. Turning on any combination of switches while pressing any number of pushbuttons will add the corresponding 4-bit numbers and light the LEDs with the result. If there is a carry (for example, turn on SW3 and press BTN3 simultaneously), the green carry LED will illuminate.

The programming process performed here stored the program in FPGA RAM. If you cycle power on the FPGA board, you will need to repeat the programming process to reload the program. Alternatively, you can store the FPGA configuration file in onboard flash memory as described in the following section.

Programming the bitstream to onboard flash memory

To configure the FPGA each time power is applied to the Arty board, the FPGA configuration file must be stored to flash memory on the board. If the **MODE** jumper is installed, the FPGA will attempt to download a configuration file from onboard flash memory at power-on. This memory is located in a separate chip adjacent to the Artix-7 FPGA. Follow these steps to program the configuration file to flash memory:

1. Install the **MODE** jumper on the Arty board if it is not already in place.
2. Right-click **Generate Bitstream** and select **Bitstream Settings....**
3. In the **Settings** dialog, check the box next to **-bin_file** and click **OK**:

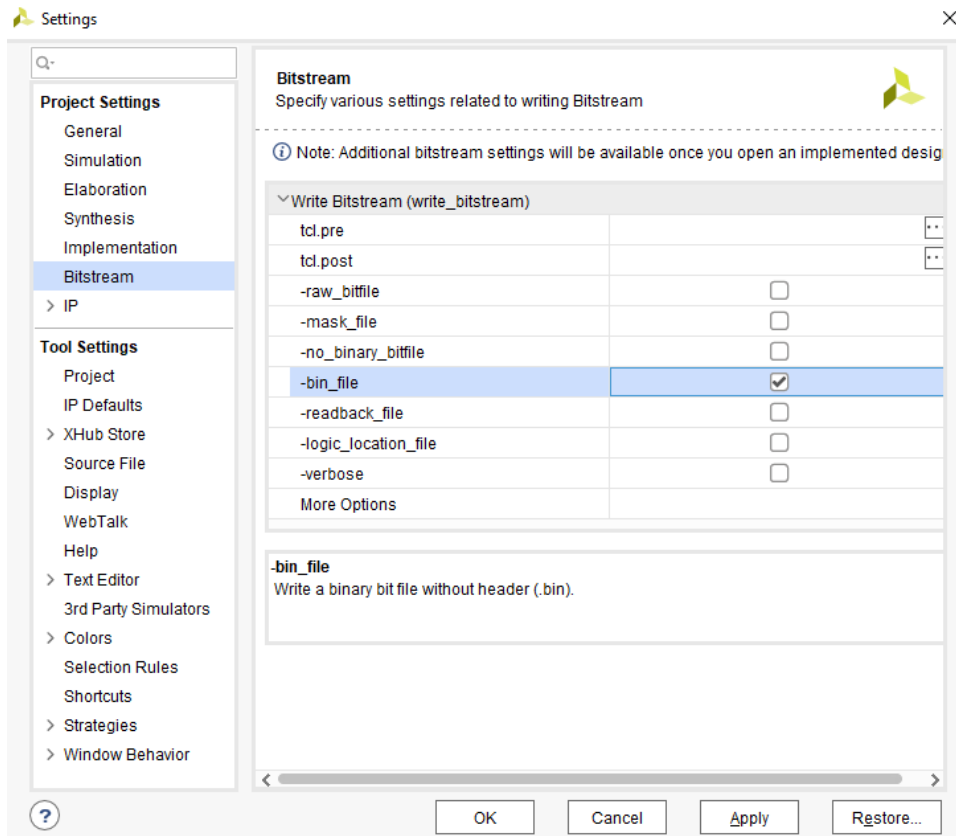


Figure 4.31 – Bitstream settings dialog

4. In the main Vivado dialog, click **Generate Bitstream** and repeat the bitstream generation process. Click **Cancel** when the **Bitstream Generation Completed** dialog appears.

5. In the **Hardware** dialog, right-click the FPGA part number (**xc7a100t_0**) and select **Add Configuration Memory Device...**:

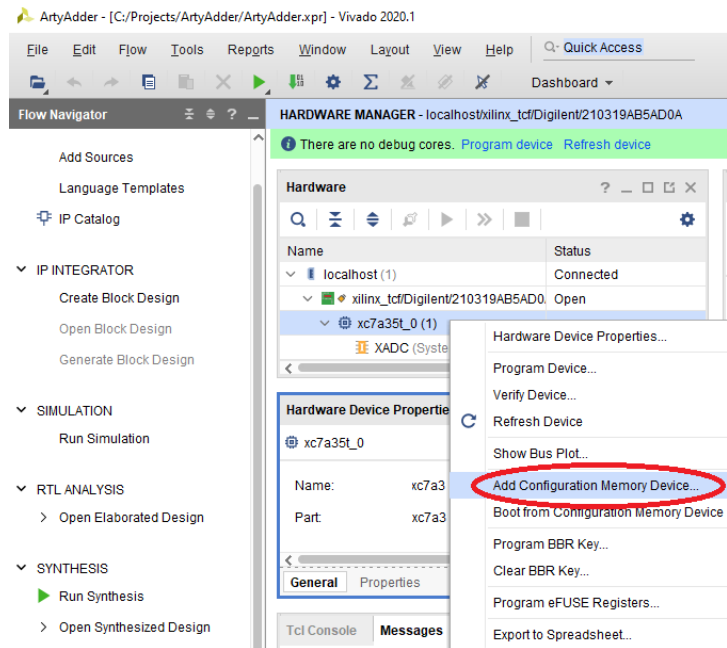


Figure 4.32 – Add Configuration Memory Device... menu selection

6. Type **s25fl127** into the **Search** box. This should bring up one matching part number. Select the part and click **OK**:

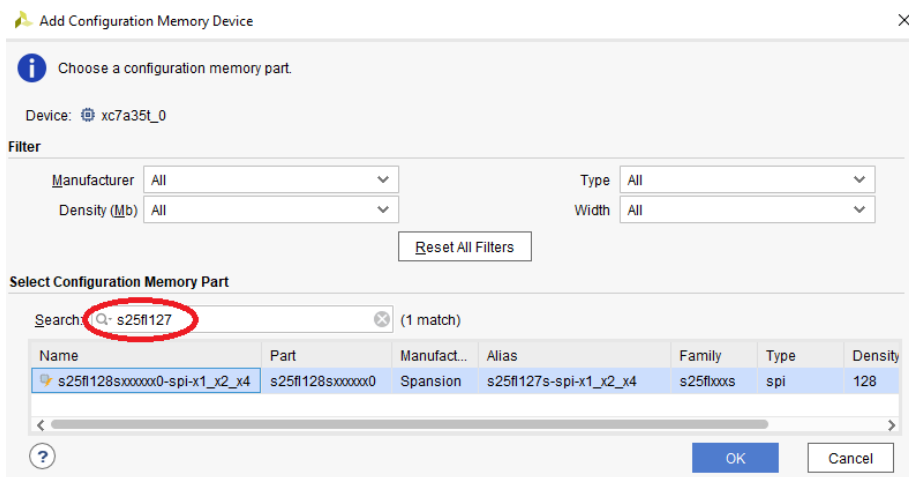


Figure 4.33 – Add Configuration Memory Device dialog

7. You will be presented with a dialog asking **Do you want to program the configuration memory device now?** Click **OK**.
8. This will bring up a **Program Configuration Memory Device** dialog requesting the configuration filename. Click the ... button to the right of **Configuration file** and select `C:/Projects/ArtyAdder/ArtyAdder.runs/impl_1/ARTY_ADDER.bin`. Click **OK**:

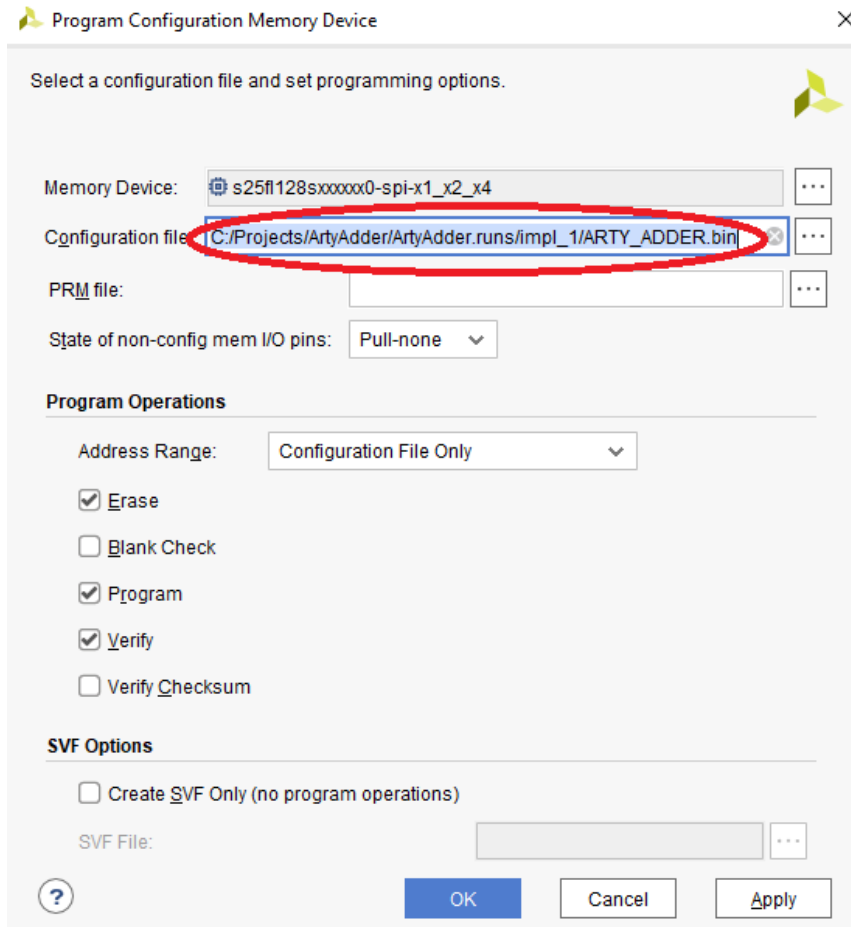


Figure 4.34 – Program Configuration Memory Device dialog

9. The programming process will take several seconds to complete. You should receive a message indicating success after the file has been programmed into the board flash memory:

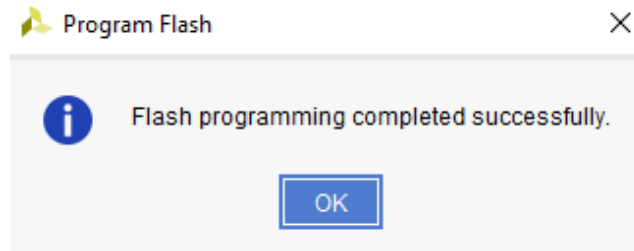


Figure 4.35 – Program Flash complete dialog

After this, each time you cycle the board power, the 4-bit adder program will load and run. It will take a long time for the program to load with the settings that we used for configuration file loading. To avoid waiting for the FPGA to load the program, you can improve the speed of configuration file loading by performing the following steps:

1. Select **Open Synthesized Design** in **Flow Navigator**.
2. In the Vivado main menu, select **Tools/Edit Device Properties...**
3. In the **General** tab, set **Enable Bitstream Compression** to **TRUE**.
4. In the **Configuration** tab, set **Configuration Rate (MHz)** to 33, then click **OK**.
5. Generate the bitstream again, and program the flash memory as described previously. You will need to remove the configuration memory device and add it back again to display the option for reprogramming.
6. Close **Hardware Manager**.
7. Unplug the Arty board USB cable and plug it in again. The program should begin running virtually instantaneously at power-on.

This section presented an example of simple combinational logic interacting with signals on the FPGA I/O pins. The intent here has been to familiarize you with the Vivado tool suite and to demonstrate how the tools are used to perform a complete FPGA development cycle.

Summary

This chapter began with a discussion on the effective use of FPGAs in real-time embedded system architectures and continued with a description of standard FPGA devices and the low-level components they contain. The range of FPGA design languages, including HDLs, block diagram methods, and popular software programming languages such as C/C++, was introduced. An outline of the FPGA development process was presented. The chapter concluded with a complete example of an FPGA development cycle, starting with a statement of requirements and ending with a functional system implemented on a low-cost FPGA development board.

Having completed this chapter, you should know how FPGAs can be applied effectively in real-time embedded system architectures and understand the components inside an FPGA integrated circuit. You have learned about the programming languages used in the design of FPGA algorithms, the steps in the FPGA development cycle, and understand the sequence of steps in the FPGA development process.

The next chapter will expand on the FPGA development process to provide a complete approach to architecting real-time embedded systems containing FPGAs. It will also begin the development of a prototype high-performance embedded system, a digital oscilloscope, that will serve as an example for the following chapters.

5

Implementing systems with FPGAs

This chapter dives into the process of designing and implementing systems with FPGAs. It begins with a description of the FPGA compilation software tools that convert a description of a logic design in a programming language into an executable FPGA configuration. We will discuss the types of algorithms best suited to FPGA implementation and suggest a decision approach for determining whether a particular embedded system algorithm is more appropriately implemented using a traditional processor or with an FPGA. The chapter ends with the step-by-step development of a baseline FPGA-based processor project that will be expanded to implement a high-speed digital oscilloscope using circuitry and software developed in later chapters.

After completing this chapter, you will have learned about the processing steps performed by FPGA compilation tools and will understand the types of algorithms best suited to FPGA implementation. You will know how to determine whether FPGA implementation is right for a given design and will have worked through a real FPGA system development project for a high-performance processing application.

We will cover the following topics in this chapter:

- The FPGA compilation process
- Algorithm types most suitable for FPGA implementation
- Kicking off the oscilloscope FPGA project

Technical requirements

We will be using Xilinx Vivado and an Arty A7-100 development board in this chapter. See *Chapter 4, Developing Your First FPGA Program*, for information on Vivado download and installation.

The files for this chapter are available at <https://github.com/PacktPublishing/Architecting-High-Performance-Embedded-Systems>.

The FPGA compilation process

The process of compiling a digital circuit model begins with a specification of circuit behavior in a hardware description language, such as VHDL or Verilog, and produces as its output an implementation of that circuit that can be downloaded and executed in an FPGA. The software tool set that performs the synthesis process is sometimes called a **silicon compiler** or **hardware compiler**.

FPGA compilation takes place in three steps: synthesis, placement, and routing. *Chapter 4, Developing Your First FPGA Program*, introduced these steps. Behind the scenes, the software tools that perform the steps implement a collection of sophisticated algorithms to produce an optimized FPGA configuration that correctly implements the circuit described by the source code.

Before you can begin the compilation process, the first step is to create a complete description of the circuit, typically as a collection of files in the VHDL or Verilog languages. This is called *design entry*.

Design entry

We will continue with the 4-bit adder circuit we used for the example project in *Chapter 4, Developing Your First FPGA Program*. The purpose of this example is to clarify that, as in traditional software development, there are multiple ways to solve a given problem in hardware description languages. As we'll see, it is generally better to choose an implementation that is clear and understandable for the developer than to try to create a more optimized but also more complicated design. We'll also introduce some new constructs in the VHDL language.

From a review of the code in `Adder4.vhdl`, listed in the *Creating VHDL source files* section of *Chapter 4, Developing Your First FPGA Program*, you will observe that the carry output from each full adder (the signal named `C_OUT`) is used as an input for the next full adder (as the signal named `C_IN`). As a result of this configuration, which is referred to as a **ripple-carry adder**, it is possible for a carry from the least significant adder to propagate through all of the higher-order adders. For example, this occurs when 1 is added to the binary value 1111. The circuitry interacting with the adder must therefore wait for this maximum propagation delay after presenting inputs to the adder before it can reliably read the adder output.

If we look at this problem at a higher level of abstraction, we can examine alternative implementations that produce the same computed result in a different manner. Instead of specifying the exact set of logic operations to perform the addition operation, in terms of AND, OR, and XOR operators, as we did in `FullAdder.vhdl`, we can instead create a **Lookup Table (LUT)** that uses an input of 8 bits, formed by concatenating the two 4-bit addends, with output consisting of the 4-bit sum and the 1-bit carry. The following listing shows how this table is represented in VHDL:

```
-- Load the standard libraries

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- Define the 4-bit adder inputs and outputs

entity ADDER4LUT is
  port (
    A4      : in    std_logic_vector(3 downto 0);
    B4      : in    std_logic_vector(3 downto 0);
    SUM4     : out   std_logic_vector(3 downto 0);
    C_OUT4  : out   std_logic
  );
end entity ADDER4LUT;

-- Define the behavior of the 4-bit adder

architecture BEHAVIORAL of ADDER4LUT is
```

```
begin
  ADDER_LUT : process (A4, B4) is
    variable concat_input : std_logic_vector(7 downto 0);
  begin
    concat_input := A4 & B4;
    case concat_input is
      when "00000000" =>
        SUM4 <= "0000"; C_OUT4 <= '0';
      when "00000001" =>
        SUM4 <= "0001"; C_OUT4 <= '0';
      when "00000010" =>
        SUM4 <= "0010"; C_OUT4 <= '0';
      .
      .
      .
      when "11111110" =>
        SUM4 <= "1101"; C_OUT4 <= '1';
      when "11111111" =>
        SUM4 <= "1110"; C_OUT4 <= '1';
      when others =>
        SUM4 <= "UUUU"; C_OUT4 <= 'U';
    end case;
  end process ADDER_LUT;
end architecture BEHAVIORAL;
```

In this listing, 250 of the 256 table entries were elided as indicated by the vertical ...

The case statement in this code is enclosed in a `process` statement. `process` statements provide a way to insert collections of sequentially executed statements within the normal VHDL construct of concurrently executing statements. The `process` statement includes a **sensitivity list** (containing A4 and B4 in this example) that identifies the signals that trigger the execution of the `process` statement whenever they change state.

Although a `process` statement contains a set of statements that execute sequentially, the `process` statement itself is a concurrent statement that executes in parallel with the other concurrent statements in the design once its execution has been triggered by a change to a signal in its sensitivity list. If the same signal is assigned different values multiple times within a process body, only the final assignment will take effect.

You may wonder why it is necessary to include the `when others` condition at the conclusion of the case statement. Although all 256 possible combinations of 0 and 1 bit values are covered in the `when` conditions, the VHDL `std_logic` data type includes representations of other signal conditions, including uninitialized inputs. The `when others` condition causes the outputs of the adder to return unknown values if any input has a value other than 0 or 1. This is a form of defensive coding that will cause a simulation run to alert us if we forget to connect any of the inputs to this logic component, or use an uninitialized or otherwise inappropriate value as an input.

These rules and behaviors are likely to be unfamiliar to anyone comfortable with traditional programming languages. When implementing and testing VHDL designs, you can expect to experience some confusing experiences where things don't seem to be functioning as you expect. You'll need to keep in mind that with VHDL you are defining digital logic that acts in parallel, not a sequentially executing algorithm. It is also important to thoroughly simulate circuit behavior and ensure your code is operating properly before attempting to run it in an FPGA.

Getting back to our example, while it might be reasonable to include a 256-element lookup table in the design of a 4-bit adder, the size of the lookup table quickly becomes unmanageable if the data words to be added consist of 16, 32, or 64 bits. Rather than using large lookup tables, practical digital adder circuits use more sophisticated logic than the ripple-carry adder, in the form of a **carry look-ahead adder**, to improve execution speed. The carry look-ahead adder includes logic to anticipate the propagation of carries, thereby reducing the time taken to produce a final result.

We will not be getting into the details of carry look-ahead adder construction, but we will note that the VHDL language contains an addition operator, and you can expect the implementation resulting from the use of this operator to be highly optimized for the targeted FPGA model. The following code is a 4-bit adder using the native VHDL addition operator rather than a collection of gate-level single-bit adders or a lookup table to perform the addition operation:

```
-- Load the standard libraries

library IEEE;
  use IEEE.STD_LOGIC_1164.ALL;
  use IEEE.NUMERIC_STD.ALL;

-- Define the 4-bit adder inputs and outputs

entity ADDER4NATIVE is
  port (
    A4      : in    std_logic_vector(3 downto 0);
    B4      : in    std_logic_vector(3 downto 0);
    SUM4     : out   std_logic_vector(3 downto 0);
    C_OUT4  : out   std_logic
  );
end entity ADDER4NATIVE;

-- Define the behavior of the 4-bit adder

architecture BEHAVIORAL of ADDER4NATIVE is

begin

  ADDER_NATIVE : process (A4, B4) is

    variable sum5 : unsigned(4 downto 0);

  begin
```

```

    sum5 := unsigned('0' & A4) + unsigned('0' & B4);

    SUM4  <= std_logic_vector(sum5(3 downto 0));
    C_OUT4 <= std_logic(sum5(4));

end process ADDER_NATIVE;

end architecture BEHAVIORAL;

```

This example includes the `IEEE.NUMERIC_STD` package, which brings in the ability to use numeric data types such as signed and unsigned integers in addition to the logic data types defined in the `IEEE.STD_LOGIC_1164` package. The example code performs type conversions from the `std_logic_vector` data type to the unsigned integer type and uses those numeric values to compute the `sum5` intermediate value, which is the 5-bit sum of the `A4` and `B4` addends. Each addend is extended from 4 to 5 bits by prepending it with a single bit of zero using the syntax `'0' & A4`. The unsigned result is converted back to the 4-bit `std_logic_vector` result `SUM4` and a single-bit `std_logic` carry output `C_OUT4`.

Summarizing this series of VHDL examples, we have seen three different implementations of the 4-bit adder circuit: first as a collection of four single-bit adder logic circuits, second as a lookup table that produces its output by simply looking up the results given the inputs, and, finally, using the native VHDL addition operator. While this has been a somewhat contrived example, it should be clear that any given algorithm can be described in a variety of ways in VHDL or other **Hardware Description Languages (HDLs)**.

With the design entry completed, we are now ready to perform logic synthesis.

Logic synthesis

FPGA circuits of moderate to high complexity typically consist of both combinational logic, which was the subject of the example project in *Chapter 4, Developing Your First FPGA Program*, and sequential logic. The output of a combinational logic circuit depends only on its inputs at a given moment, as we saw with various combinations of switch and pushbutton inputs on the Arty A7 board as it performed the addition operation.

Sequential circuits, on the other hand, maintain state information representing the results of past operations that affects future operations. Sequential logic within a circuit, or within a functional subset of a circuit, almost always uses a shared clock signal to trigger the update of coordinated data storage elements. By using a clock signal, these updates occur simultaneously and at regular intervals defined by the clock frequency. Sequential logic circuits that update state information based on a common clock signal are referred to as **synchronous sequential logic**. Most sophisticated digital circuits can be represented as hierarchical arrangements of lower-level components consisting of combinational and synchronous sequential logic.

FPGA devices generally implement combinational logic using lookup tables, which represent the output of a logic gate configuration using a small RAM. For a typical 6-input lookup table, the RAM contains Instance 4 single-bit entries, where each entry contains the single-bit circuit output for one possible combination of input values. *Figure 5.1* shows the inputs and output of this simple lookup table:

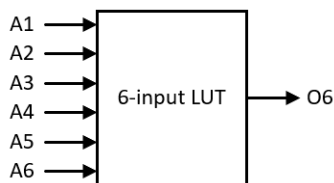


Figure 5.1 – 6-input lookup table

More complex combinational circuits can be constructed by combining multiple lookup tables in parallel and in series. The synthesis tool performs this step automatically for you.

FPGAs use flip-flops, block RAM, and distributed RAM to hold state information. Lookup tables and the components containing state information form the raw materials from which complex circuit designs are constructed by the silicon compiler.

Clock signals drive the operation of synchronous sequential logic within the digital circuit. In a typical FPGA design, a number of clock signals will be defined with frequencies that vary depending on the function that uses each signal. Clock signals internal to the FPGA may drive high-speed operations at frequencies in the hundreds of MHz. Other clock signals that drive interfaces to external peripherals, such as an Ethernet interface or DDR3 RAM, may have frequencies tailored to the needs of the external hardware. FPGAs generally contain clock generation hardware that supports the production of several clock frequencies for various uses.

Using vendor-provided FPGA development tools, system designers define the logic circuit, typically in VHDL or Verilog, and provide information describing the circuit clocking requirements and constraints associated with I/O interfaces and timing. The compilation tools provided by the vendor then perform the steps of synthesis, placement, and routing.

A significant portion of the processing effort involved in FPGA synthesis focuses on minimizing the amount of FPGA resources consumed by the implementation while meeting timing constraints. By minimizing resource consumption, the tools enable more complex designs to be implemented in smaller, less expensive devices. The next section will discuss some aspects of the design optimization process.

Design optimization

A significant portion of the processing that occurs during synthesis, implementation, and routing is devoted to optimizing the performance of the circuit. This optimization process has multiple goals, including minimizing resource usage, achieving maximum performance (in terms of maximum clock speed and minimum propagation delay), and minimizing power consumption.

By using an appropriate selection of constraints, discussed later in this chapter, it is possible for the designer to focus the optimization process on the goals most appropriate for the system under development. For example, a battery-powered system may put a premium on power consumption while being less concerned with achieving peak performance.

The optimization performed by these tools can be quite surprising to users unfamiliar with their capabilities. It is important to understand that the tools are not limited to implementing a circuit that resembles the way you have laid it out in VHDL code. The only requirement the tools must abide by is ensuring that the implemented circuit functions identically, in terms of input and output, to the design described in the code.

To emphasize this point, we can look at the performance (in terms of maximum propagation delay), resource utilization (in terms of the number of lookup tables used), and power consumption in milliwatts to compare the three forms of the 4-bit adder circuit design we developed in the earlier examples. We will continue with the 4-bit adder using the native VHDL addition operator listed earlier in this chapter.

Before we can do this, we first need to add two lines to the end of the `Arty-A7-100.xdc` constraints file:

```
create_clock -period 10 -name virtual_clock
set_max_delay 12.0 -from [all_inputs] -to [all_outputs]
```

The `create_clock` statement creates a virtual clock, which we have named `virtual_clock`, with a period of 10 ns. This is necessary because our circuit does not have any actual clock signals, but Vivado requires a reference clock to perform timing analysis, even if the clock is not really present. The `set_max_delay` statement defines a constraint stating that the maximum propagation delay we can tolerate for our FPGA implementation is 12 ns between any input and any output. We do not have a real need for propagation to take no more than 12 ns. We have chosen this limit because it is within the capabilities of the FPGA device.

Perform the following steps to implement the design:

1. Save the preceding changes to the XDC file.
2. Select **Run Implementation**. You will be prompted to rerun the synthesis step, which is necessary to incorporate the changed constraints.
3. When implementation completes, select **Open Implemented Design**.
4. Select **Timing Analysis** in the drop-down list near the top-right corner of the main window.
5. Click **Project Summary** and maximize the window.
6. Scroll to the bottom, if necessary, to locate the **Timing**, **Utilization**, and **Power** summaries.

The following figure highlights key items on the **Timing**, **Utilization**, and **Power** summaries:

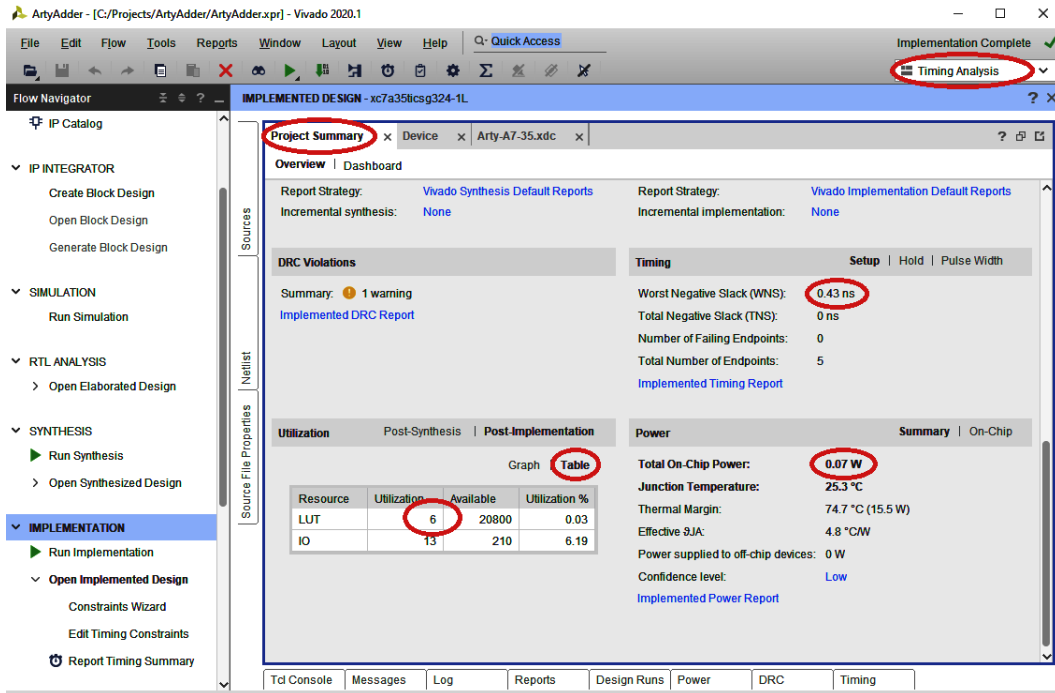


Figure 5.2 – The Timing, Utilization, and Power summaries

This display shows a **Worst Negative Slack (WNS)** time of 0.43 ns. Numerically, this value represents the most marginal propagation path in the design relative to our maximum delay constraint of 12 ns. Because the WNS is positive, all timing constraints have been met. The actual worst-case propagation delay in the circuit is our constraint (12 ns) minus WNS (0.43 ns), which is 11.57 ns.

The **Utilization** summary displays the FPGA resource consumption in terms of lookup tables and I/O pins. Click **Table** to display the number of each item used. Our design consumes **6** LUTs and uses **13** I/O pins (4 switches, 4 buttons, 4 green LEDs, and 1 multicolor LED). The **Power** summary indicates that our design consumes **0.07 W** of power.

If you haven't done so already, add design source files named `Adder4LUT.vhdl` and `Adder4Native.vhdl` to your project. Insert the 4-bit adder definition into each file that contains the source code for each model as described earlier in this chapter. For the `Adder4LUT.vhdl` model, because the code in the `case` statement is so lengthy and repetitive, you should use your favorite programming language to generate the textual contents of the `case` statement.

You can switch your implementation of the adder between these three designs by changing two lines in the `ArtyAdder.vhdl` file. The following example code shows the architecture section of `ArtyAdder.vhdl` with the two lines changed to select `ADDER4NATIVE` as the 4-bit adder implementation:

```
architecture BEHAVIORAL of ARTY_ADDER is

    -- Reference the previous definition of the 4-bit adder

    component ADDER4NATIVE is
        port (
            A4      : in    std_logic_vector(3 downto 0);
            B4      : in    std_logic_vector(3 downto 0);
            SUM4     : out   std_logic_vector(3 downto 0);
            C_OUT4  : out   std_logic
        );
    end component;

begin

    ADDER : ADDER4NATIVE
        port map (
            A4      => sw,
            B4      => btn,
            SUM4     => led,
            C_OUT4  => led0_g
        );

end architecture BEHAVIORAL;
```

After changing those two lines, rerun the synthesis and implementation, then view the **Timing Utilization** and **Power** summaries. The results of this procedure for each of the three adder designs are shown in the following table:

Adder type	Max delay (ns)	LUT utilization	Power consumption (W)
Adder4	11.708	6	0.07
Adder4Native	11.57	6	0.07
Adder4LUT	11.638	7	0.07

From these results, it is clear that, although these three design variants were based on substantially different approaches to defining the 4-bit addition operation, and they required significantly varying amounts of source code, the optimized form of each resulted in very similar FPGA implementations in terms of performance and resource utilization.

The key takeaway from this analysis is that developers should not exert substantial effort in attempting to lay out an overly complex, fully optimized design. Instead, create a design that is understandable, maintainable, and functionally correct. Leave the optimization to the compilation tools. They are very good at it.

The next section introduces an alternative method for design entry: high-level synthesis.

High-level synthesis

Our examples up to this point have consisted of VHDL code-based circuit designs. As discussed in the *FPGA implementation languages* section of *Chapter 4, Developing Your First FPGA Program*, it is also possible to implement FPGA designs using traditional programming languages such as C and C++.

Although other languages are available for use in high-level synthesis, our discussion will focus on C and C++. The high-level synthesis tools from Xilinx are available within an integrated development environment named *Vitis HLS*. If you installed Vitis along with Vivado, you can start Vitis HLS by double-clicking the icon with that name on your desktop.

Although most of the capabilities of the C and C++ languages are available in Vitis HLS, there are some significant limitations you must keep in mind:

- Dynamic memory allocation is not allowed. All data items must be allocated as automatic (stack-based) values or as static data. All features of C and C++ that rely on the use of heap memory are unavailable.
- Library functions that assume the presence of an operating system are not available. There is no file reading or writing, and no interaction with users via the console.
- Recursive function calls are not allowed.
- Some forms of pointer conversion are prohibited. Function pointers are not allowed.

One important feature that is available in Vitis HLS that is not available natively in standard C/C++ is support for arbitrary precision integers and for fixed-point numbers. Fixed-point numbers are integers that can represent fractional values by placing the decimal point at a fixed location within the data bits. For example, a 16-bit fixed-point number with the decimal point placed before the two least significant bits has a fractional resolution of $\frac{1}{4}$. In this format, the number 4642.25 is represented by the binary value 01001000100010.01.

To gain some familiarity with high-level synthesis, let's implement our 4-bit adder in C++ within Vitis HLS. To do so, follow these steps:

1. Double-click the Vitis HLS icon to start the application.
2. Click **Create Project** in the Vitis main dialog.
3. Name the project `ArtyAdder4HLS` and set the location to `C:\Projects` as shown in the following figure. Click **Next**:

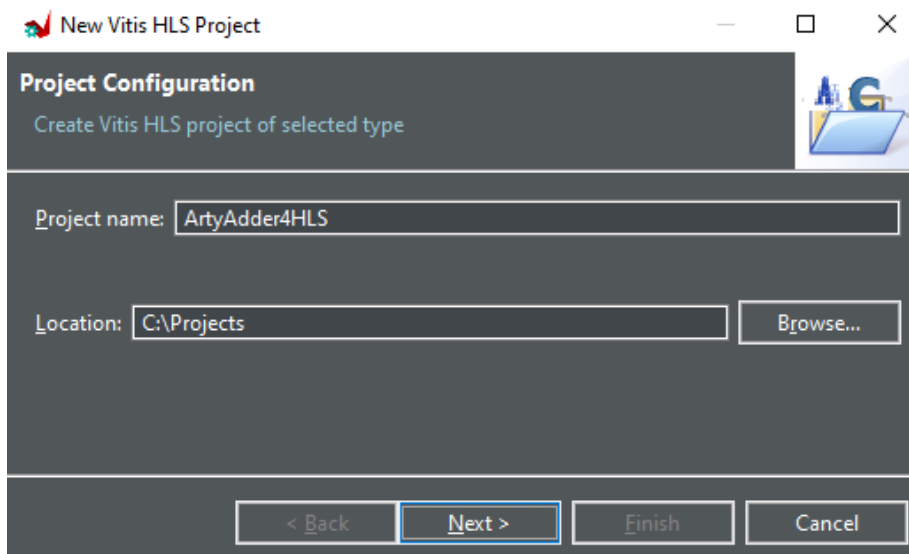


Figure 5.3 – Vitis HLS project configuration dialog

4. Set **Top Function** to `ArtyAdder4HLS` as shown in the following figure, then click **Next**:

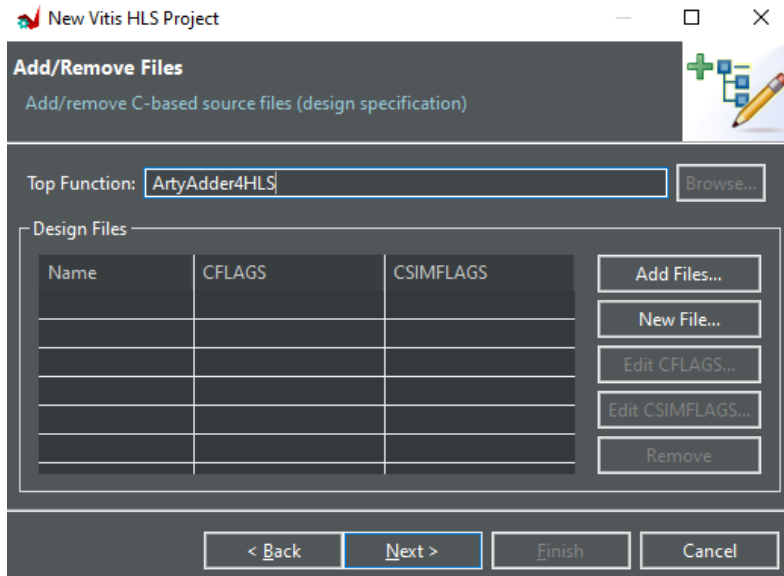


Figure 5.4 – Vitis HLS Add/Remove Files dialog

5. Click **Next** to skip past the **Add/remove C-based testbench files** dialog.
6. Set **Solution Name** to **ArtyAdder4HLS** as shown in the following figure, then click **Finish**:

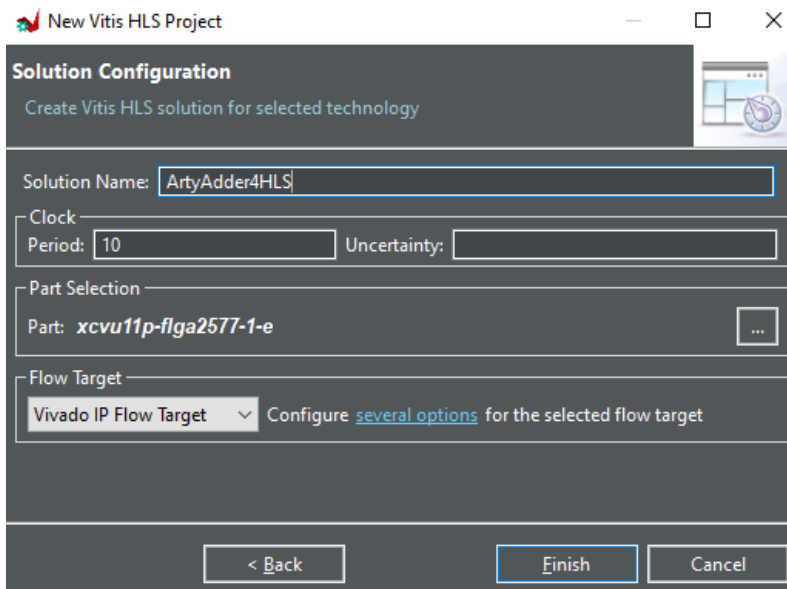


Figure 5.5 – Vitis HLS solution configuration dialog

7. In the Vitis **Explorer** sub-window, right-click **Source** and select **New File...**. Enter the name `ArtyAdder4HLS.cpp` and store it at `C:\Projects\ArtyAdder4HLS`.
8. Insert the following code into the `ArtyAdder4HLS.cpp` file:

```
#include <ap_int.h>

void ArtyAdder4HLS(ap_uint<4> a, ap_uint<4> b,
                  ap_uint<4> *sum, ap_uint<1> *c_out)
{
    unsigned sum5 = a + b;

    *sum = sum5;
    *c_out = sum5 >> 4;
}
```

9. The data types identified as `ap_uint<>` are arbitrary-precision unsigned integers with the number of bits indicated between the angle brackets. Observe that the C++ statements in the body of the function closely mirror the statements in the `Adder4Native` version of the 4-bit adder.
10. Click the green triangle in the icon ribbon to start the synthesis process. If prompted, answer **Yes**, you want to save the editor file.

Vitis HLS will generate both Verilog and VHDL versions of the model. Expand the `ArtyAdder4HLS` folder in the Vitis Explorer and double-click the `ArtyAdder4HLS.vhd` file to open it in the editor as shown in the following screenshot:

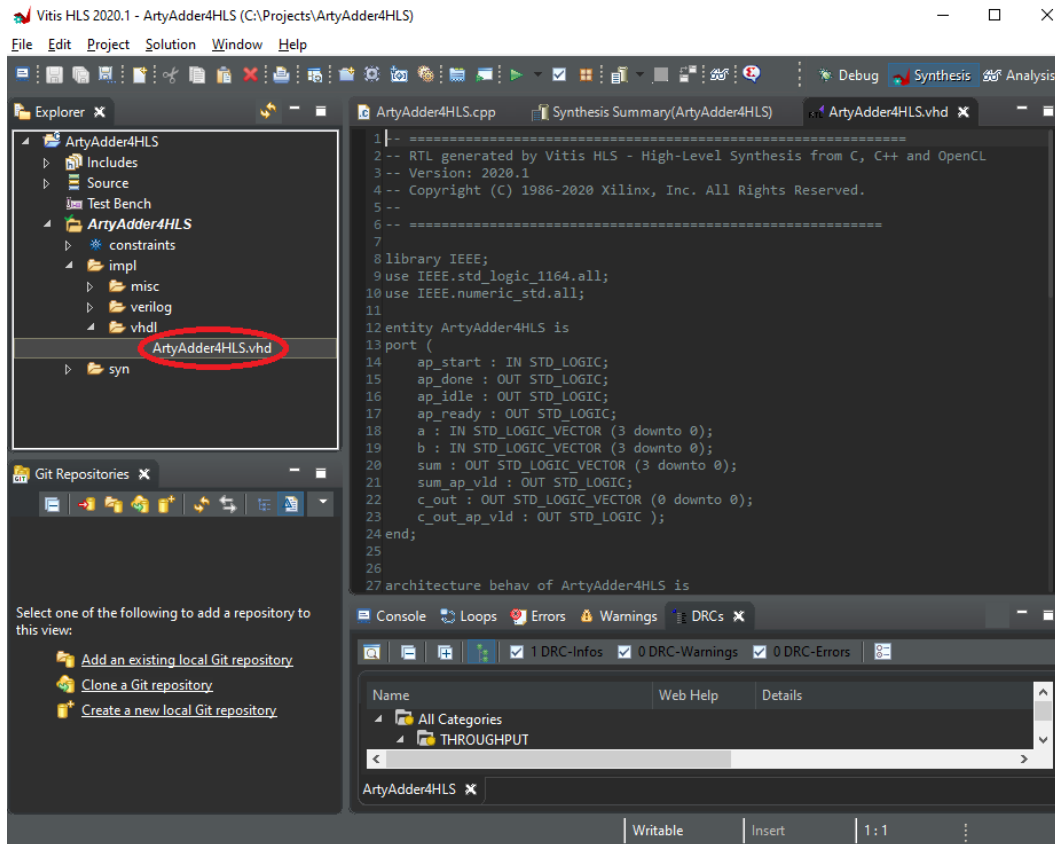


Figure 5.6 – ArtyAdder4HLS.vhd file contents

Vitis HLS adds a control input (`ap_start`) and status outputs (`ap_done`, `ap_idle`, `sum_ap_vld`, and `c_out_ap_vld`) to the list of inputs and outputs we have defined for the 4-bit adder. Those signals are not relevant for our circuit, which contains purely combinational logic.

Copy the `ArtyAdder4HLS.vhd` file to the folder containing the VHDL files for our ArtyAdder project (`C:\Projects\ArtyAdder\ArtyAdder.srcs\sources_1\new`). Add that file to the project and create a new file named `ArtyAdder4HLSWrapper.vhdl` with the following content:

```

-- Load the standard libraries

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

```



```
use IEEE.NUMERIC_STD.ALL;

-- Define the 4-bit adder inputs and outputs

entity ADDER4HLSWRAPPER is
  port (
    A4      : in      std_logic_vector(3 downto 0);
    B4      : in      std_logic_vector(3 downto 0);
    SUM4     : out     std_logic_vector(3 downto 0);
    C_OUT4  : out     std_logic
  );
end entity ADDER4HLSWRAPPER;

-- Define the behavior of the 4-bit adder

architecture BEHAVIORAL of ADDER4HLSWRAPPER is

  component ARTYADDER4HLS is
    port (
      AP_START      : in      std_logic;
      AP_DONE       : out     std_logic;
      AP_IDLE       : out     std_logic;
      AP_READY      : out     std_logic;
      A              : in      std_logic_vector(3 downto 0);
      B              : in      std_logic_vector(3 downto 0);
      SUM            : out     std_logic_vector(3 downto 0);
      SUM_AP_VLD    : out     std_logic;
      C_OUT          : out     std_logic_vector(0 downto 0);
      C_OUT_AP_VLD  : out     std_logic
    );
  end component;

  signal c_out_vec : std_logic_vector(0 downto 0);

begin
```

```

-- The carry input to the first adder is set to 0
ARTYADDER4HLS_INSTANCE : ARTYADDER4HLS
  port map (
    AP_START      => '1',
    AP_DONE       => open,
    AP_IDLE       => open,
    AP_READY      => open,
    A             => A4,
    B             => B4,
    SUM           => SUM4,
    SUM_AP_VLD    => open,
    C_OUT         => c_out_vec,
    C_OUT_AP_VLD  => open
  );

C_OUT4 <= c_out_vec(0);

end architecture BEHAVIORAL;

```

Here, we have set the unused input to the ARTYADDER4HLS_INSTANCE component to 1 and set the unused outputs to open, meaning the signals are unconnected.

After completing the implementation of this design, we can augment the table in the *Design optimization* section to add a row with the parameters for the C++ HLS version of the 4-bit adder:

Adder type	Max delay (ns)	LUT utilization	Power consumption (W)
Adder4	11.708	6	0.07
Adder4Native	11.57	6	0.07
Adder4LUT	11.638	7	0.07
Adder4HLS	11.485	6	0.07

We see that the C++ version of the 4-bit adder results in the best performance in terms of minimizing propagation delay and minimizing resource usage. This may be surprising if you had assumed that coding directly in HDL would produce a better performing result. Of course, this is a very simple example and it would not be prudent to assume similar results would apply to a very complex design. But they might!

Optimization and constraints

In general, given the large quantity of resources available in a particular model of FPGA, it is possible to implement a particular logic circuit source code definition in a tremendous variety of ways. Although each of these variations will perform the logical functions of the circuit in an identical manner, each implementation will be unique in various aspects. Some configurations will use more of the FPGA resources and some will use less. Some will be faster, in terms of propagation delay and attainable clock speed, and some will be slower. Some will consume more power, and some will consume less. Of all the possible ways a specific circuit might be implemented in a given FPGA, how should the tool select the configuration to implement?

By default, the Vivado tools place the highest priority on optimizing timing by minimizing the propagation time of signals that have the slowest paths. As secondary optimization goals, the area (in terms of FPGA resource usage) will be minimized and power consumption will be minimized. For advanced users, configuration options are available to adjust the relative priority of the optimization goals and to adjust the amount of effort, in terms of execution time, to put into the search for an optimal design. Given the very large number of possible configurations, it is not possible for the tools to evaluate every one of them. In general, the result of the optimization process is a design that may not be the best possible configuration. Instead, the result is a design that meets specifications and is likely to be not very far from the absolute best design in terms of performance metrics.

Practical FPGA designs require constraints on the optimization process. One obvious category of constraints is the selection of I/O pins for signals. By indicating that a particular signal must be connected to a specific I/O pin, the synthesis and implementation processes must then restrict their search to consider only configurations that have that signal connected to the given pin. Every I/O signal used by the FPGA logic must have an I/O pin constraint. These constraints define the interface between the FPGA and external circuitry.

The other primary category of constraints relates to timing. In FPGA designs that implement synchronous sequential logic, proper operation of the circuitry is dependent on the reliable operation of flip-flop-based registers. Flip-flops are clocked devices, meaning they capture the input signal on a clock edge. For the input data to be captured reliably, the input signal must be stable for some period of time before the clock edge and must remain stable for an additional length of time following the clock edge.

Figure 5.7 shows a simplified version of the D flip-flop discussed in *Chapter 1, Architecting High-Performance Embedded Systems*, along with a timing diagram of the **D** and **Clock** inputs to the flip-flop:

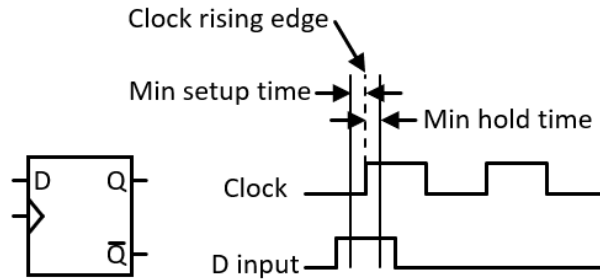


Figure 5.7 – D flip-flop input timing constraints

The flip-flop reads the **D** signal on the rising edge of the clock. For the flip-flop to read the signal reliably, the input must be at the desired level at least the minimum setup time before the clock edge and it must be held at the same level at least the minimum hold time following the clock edge. In the figure, the **D** input satisfies the timing constraints and will load a high (binary 1) value into the flip-flop on the first clock rising edge and will load a 0 on the second rising edge.

During the optimization process, the synthesis and implementation tools evaluate the setup and hold time requirements for all synchronous logic elements and attempt to produce a design that satisfies the timing requirements for all devices in the circuit.

If the design is to be connected to an external device with strict timing requirements, or if you know of particular timing constraints internal to the FPGA that the automated tools may not satisfy by default, it is possible to define additional timing constraints to meet the requirements. Basic timing constraints include the clock frequencies used by the FPGA circuit and the setup and hold times of I/O connections to the FPGA. More advanced uses of timing constraints can be employed to declare requirements related to internal communication paths within the FPGA.

An effective FPGA design contains constraint definitions for all of the I/O signal pins in use, including the timing requirements associated with those pins. The constraint set also includes internal timing requirements for the circuit such as clock frequencies used in various portions of the circuit and any other specific timing goals the circuit must satisfy. This information is used by the synthesis and implementation tools in their effort to produce a near-optimal circuit design that satisfies all of the constraints.

With this understanding of the FPGA development process, one of the first things a system architect must consider when developing a new system is whether the use of an FPGA solution even makes sense for that project. To assist with this decision process, the next section describes the types of algorithms that are most suitable for use in FPGA implementations.

Algorithm types most suitable for FPGA implementation

A key differentiating factor of an algorithm suitable for an FPGA-based solution is that data arrives faster than a standard processor, even a high-speed device, can receive the data, perform the necessary processing, and write output to the intended destination. If this is the case for a particular system architecture, the next question to ask is if there is an available off-the-shelf solution that supports the required data rate and is capable of performing the necessary processing. If no such acceptable solution exists, it is prudent to explore the use of an FPGA in the design. The following sections identify some categories of processing algorithms that often involve the use of FPGAs.

Algorithms that process high-speed data streams

Video is an example of a high-speed data source, with high-resolution video arriving at rates of tens of gigabits per second. If your application involves standard video operations such as signal enhancement, frame rate conversion, or motion compensation, it may make sense to use an existing solution rather than developing your own. But if an off-the-shelf video signal processor does not meet your needs, you may want to consider implementing your solution with a custom FPGA design.

Another category of high-speed data stream is produced by high-speed ADCs, with bit rates reaching into the gigabits per second. These ADCs are used in systems such as radar and radio communication systems, among many others. Ordinary processors cannot handle the sheer quantity of data produced by these devices, necessitating the use of an FPGA or another gate array device to perform the initial data reception and processing stages, ultimately generating a lower data rate output for consumption by a processor.

The general approach for using an FPGA in a high-speed data system requires the processing performed by the FPGA to handle the highest-speed aspects of system operation while interaction with the system processor and other peripherals takes place at substantially lower data rates.

Parallel algorithms

Computational algorithms with a high degree of parallelism can be substantially accelerated by execution on an FPGA. The naturally parallel nature of HDLs combines with high-level synthesis capabilities to provide a straightforward path for execution speed acceleration.

Some examples of parallel algorithms that may be suitable for FPGA acceleration include sorting large datasets, matrix operations, genetic algorithms, and neural network algorithms.

If you have an existing software algorithm containing parallel features, it may be possible to generate an FPGA implementation with substantially improved performance by compiling the code using high-level synthesis tools. This approach requires that the algorithm is in a programming language supported by the high-level synthesis tools. A complete system design might implement the accelerated FPGA-based algorithm as a coprocessor running alongside a standard processor that performs all of the remaining work of the system.

Algorithms using nonstandard data sizes

Processors typically operate natively on data sizes of 8, 16, 32, and sometimes 64 bits. When working with devices that produce or receive data in other sizes, for example, a 12-bit ADC, it is common to select the next larger supported data size (in this case, 16 bits) and simply ignore the extra bits appended to the actual data bits.

While this approach is acceptable for many purposes, it also results in the waste of 25% of memory storage space and communication bandwidth when working with these data values unless additional steps are taken to break up the data values and store them in system-supported 8-bit chunks.

Wouldn't it be nice if you could declare variables with a 12-bit data type natively in the software that runs on your system processor? You can't do that in general, but you can define a 12-bit data type in your FPGA model and use that type for data storage, transfer, and mathematical operations. Variables of this data type will only need to be organized into 8-bit chunks, or some multiple of 8 bits, when communicating with external devices.

This section summarized some of the types of algorithms that are candidates for acceleration by implementation in an FPGA. The examples mentioned here are by no means exhaustive. A system designer should consider the use of an FPGA in any scenario where computational throughput is a bottleneck.

In the next section, we will take all of the knowledge we have gained on the use of FPGAs in high-performance embedded systems and focus on a specific project that puts this knowledge to use: a high-speed, high-resolution, digital oscilloscope.

In this section, we will roll up our sleeves and get to work on an FPGA design project that will require the use of the FPGA development process discussed to this point, as well as a high-speed circuit board design, which we will get started on in the next chapter.

Project description

This project will develop a digital oscilloscope based on the Arty A7-100T board that uses a standard oscilloscope probe to measure voltages on a system under test. The key requirements of this project are as follows:

- The input voltage range is $\pm 10\text{V}$ when using a scope probe set to the 1X range.
- The input voltage is sampled at 100 MHz with 14 bits of resolution.
- Input triggering is based on the input signal rising or falling edge and trigger voltage level. Pulse length triggering is also supported.
- Once triggered, up to 248 MB of sequential sample data can be captured. This data will be transferred to the host PC for display after each capture sequence completes.
- The hardware consists of a small add-on board that we will design and construct in the following chapters, which will be plugged into the connectors on the Arty A7-100T development board. The Arty provides the FPGA device used in the system. The add-on board contains the ADC, oscilloscope probe input connector, and analog signal processing.
- After each sample sequence is collected, the Arty transfers the collected data to a host system over Ethernet.

This may seem to be an unusual combination of features, so I will try to provide a rationale for these requirements. The primary purpose of this example is to demonstrate architecture and development techniques for high-performance FPGA-based solutions, not to produce a product that sells well. If you work through and understand the processes used to develop this system, you should be able to move on to other high-performance FPGA designs with ease.

The oscilloscope sampling speed (100 MHz) is not especially fast for a digital oscilloscope. The reason for keeping the speed of ADC sampling at this level is, first, to keep the cost of the parts down. Extremely high-speed ADCs are very costly. The ADC used for this design costs about \$37. Second, high-speed circuit design is difficult. Extremely high-speed circuit design is even more difficult. We are only trying to provide an introduction to the issues associated with high-speed circuit design with this project, so limiting the maximum circuit frequency to a reasonable level will help prevent frustration.

The use of the Ethernet communication mechanism opens this architecture up for use as an IoT device. Most digital oscilloscopes use USB to connect to a physically nearby host. With our Ethernet connection, the oscilloscope and its user interface could potentially be on opposite sides of the earth.

The remainder of this chapter sets up a baseline Vivado design for this project.

Baseline Vivado project

This project will use a Xilinx MicroBlaze soft processor running the FreeRTOS real-time operating system to perform TCP/IP communications over the Ethernet port on the Arty A7-100. To complete this phase of the project, you will need the following items:

- An Arty A7-100T board
- A USB cable connecting the Arty board to your computer
- An Ethernet cable connecting the Arty board to your local network
- Vivado installed on your computer

I am assuming you are now a bit familiar with Vivado. Screenshots will only be provided to demonstrate features that have not been seen as part of previous examples.

The following bullet points provide an overview of the steps we will perform:

- Create a new Vivado project.
- Create a block diagram-based representation of a MicroBlaze microcontroller system with interfaces to the following components on the Arty A7 board: DDR3 SDRAM, an Ethernet interface, 4 LEDs, 4 pushbuttons, 4 RGB LEDs, 4 switches, the SPI connector (J6 on the Arty A7 board), and the USB UART.
- Define a 25 MHz clock as the reference clock for the Ethernet interface. This clock signal is assigned to the appropriate pin on the FPGA package using a constraint.
- Generate a bitstream from the design.

- Export the project from Vivado into the Vitis software development environment.
- Create a Vitis project that implements a simple TCP echo server.
- Run the software on the Arty board and observe messages sent via the UART.
- Use Telnet to verify the TCP echo server is working.

Let's get started:

1. Begin by creating a project using the steps listed in the *Creating a project* section in *Chapter 4, Developing Your First FPGA Program*. The suggested name for this project is `oscilloscope-fpga` and the location is `C:\Projects\oscilloscope-fpga`.
2. Click **Create Block Design** to open the block design window. You will be prompted for a design name. The default, **design_1**, is acceptable:

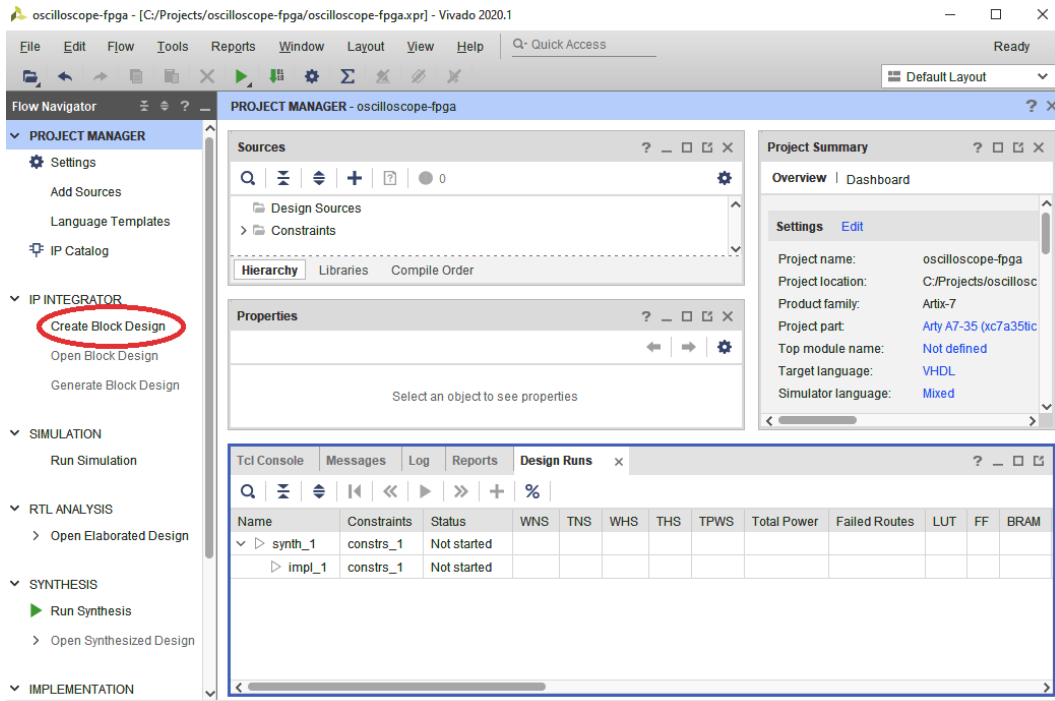


Figure 5.8 – Creating the block design

3. Select the **Board** tab in the **Block Design** window. Drag **System Clock** onto the **Diagram** window.

4. Double-click the **Clocking Wizard** component to open its **Re-customize IP** dialog. Be sure to double-click the background of the component and not on one of the pin names.
5. Select the **Output Clocks** tab in the dialog. Check the checkboxes next to **clk_out2** and **clk_out3** (**clk_out1** should already be checked). Set **Output Freq** for **clk_out1** to 166.66667 MHz, **clk_out2** to 200 MHz, and **clk_out3** to 25 MHz:

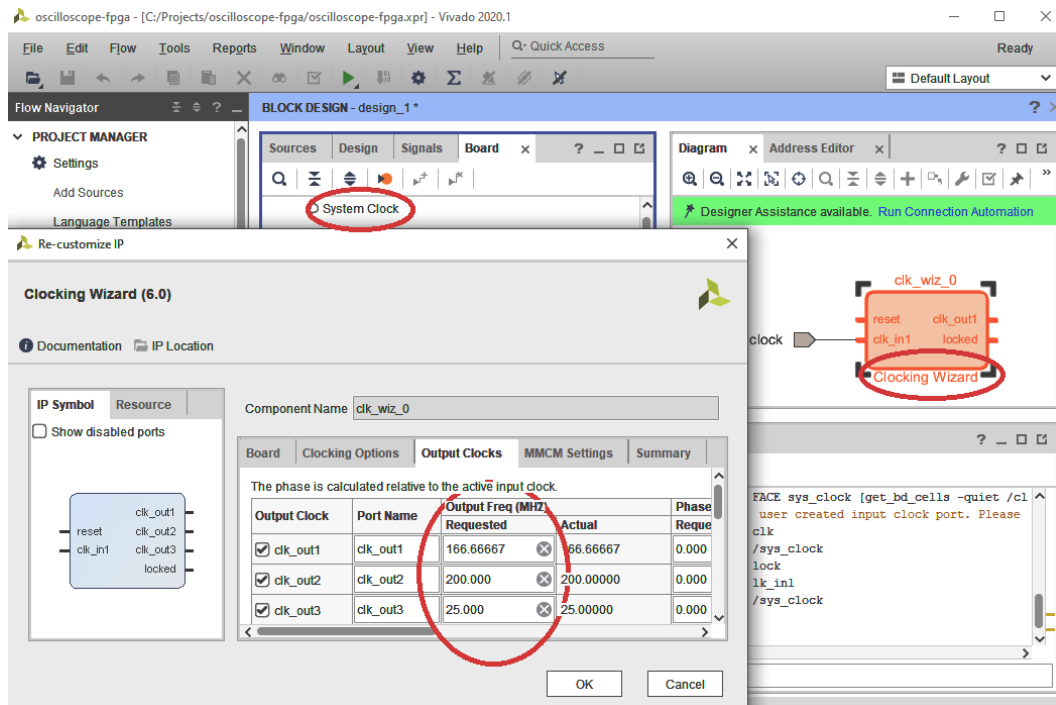


Figure 5.9 – Configuring the Clocking Wizard

6. Scroll the **Output Clocks** window to the bottom and set **Reset Type** to **Active Low**. Click **OK**.
7. On the **Clocking Wizard** component, right-click **clk_out3** and select **Make External** from the menu. A port will appear on the diagram.
8. Click on the text **clk_out3_0** in the **Diagram** window. In the **External Port Properties** window, rename **clk_out3_0** to **eth_ref_clk**.
9. Click **Run Connection Automation** in the green bar. Click **OK** in the dialog that appears.
10. Drag **DDR3 SDRAM** from the **Board** tab into the **Diagram** window.

11. Delete the **clk_ref_i** and **sys_clk_i** external ports from the **Memory Interface Generator** (click to select each port, then press the *Delete* key).
12. Click and drag from the **clk_out1** pin on **Clocking Wizard** to the **sys_clk_i** pin on the **Memory Interface Generator** to create a connecting wire.
13. Click and drag from the **clk_out2** pin on the **Clocking Wizard** to the **clk_ref_i** pin on the **Memory Interface Generator** to create a connecting wire.
14. Place the mouse over the **reset** port connected to the **Clocking Wizard** and when it appears as a pencil, click and drag to the **sys_rst** input on the **Memory Interface Generator**. This will create another connecting wire. After this step, the diagram should appear as follows:

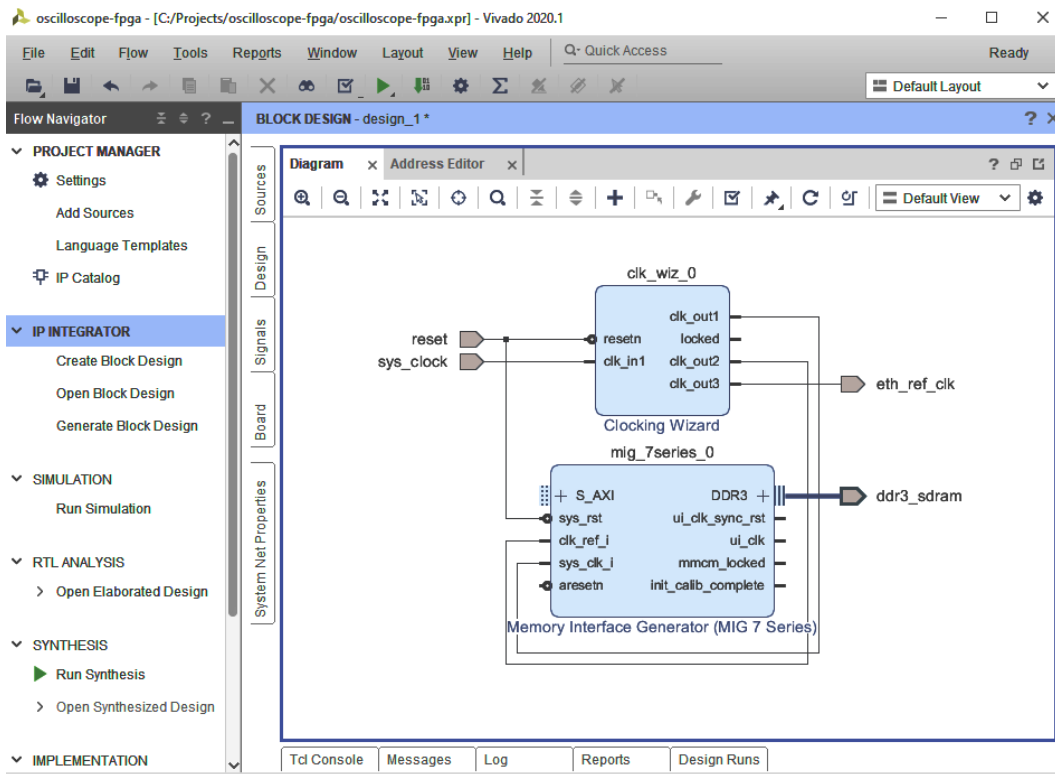


Figure 5.10 – The completed clocking configuration

15. Click the + icon at the top of the **Diagram** window and type **micro** in the search box that appears. Select the **MicroBlaze** entry and press *Enter*.

16. Click **Run Block Automation** in the green bar. Select **Real-time** as the **Preset**, set **Local Memory** to 32 KB, and set **Clock Connection** to `/mig_7series0/ui_clk (83 MHz)`. Click **OK**.
17. Click **Run Connection Automation** in the green bar. In the dialog that appears, check the box next to **All Automation** and click **OK**.
18. Double-click the **MicroBlaze** component in the diagram. Step through the numbered pages in the dialog and make the following changes: On **Page 2**, uncheck **Enable Integer Divider** and **Enable Additional Machine Status Register Instructions**, and check **Enable Branch Target Cache**. On **Page 3**, do not make any changes. On **Page 4**, set the **Instruction** and **Data Cache** sizes both to 32 kB. On **Page 5**, set **Number of PC Breakpoints** to 6. No changes are needed on **Page 6**. Click **OK**.
19. Drag the following items from the **Board** window into the **Diagram** window and click **OK** after adding each one: **Ethernet MII**, **4 LEDS**, **4 Push Buttons**, **4 RGB LEDS**, **4 Switches**, **SPI connector J6**, and **USB UART**.
20. Click the + icon at the top of the **Diagram** window and type `timer` in the search box that appears. Select the **AXI Timer** entry and press **Enter**.
21. Click **Run Connection Automation** in the green bar. In the dialog that appears, check the box next to **All Automation**. Click **OK**.
22. Find the **Concat** block on the diagram and double-click it to open the **Re-customize IP** dialog. Change **Number of Ports** to 3 and click **OK**.
23. Connect the **In0-In2** ports of the **Concat** block to the following pins, in order: **AXI EthernetLite/ip2intc_irpt**, **AXI UartLite/interrupt**, and **AXI Timer/interrupt**.
24. Press **Ctrl + S** to save the design.
25. Press **F6** to validate the design. Ensure validation is successful with no errors.
26. Select the **Sources** tab in the **Block Design** window. Right-click on **design_1** under **Design Sources** and select **Create HDL Wrapper**. Click **OK**.

This completes the block diagram for the initial phase of the project. In the next series of steps, we will add constraints to specify the characteristics of the Ethernet interface clock output pin:

1. Still in the **Sources** tab, expand **Constraints**, then right-click on **constrs_1** and select **Add Sources**.

2. In the **Add Sources** dialog, click **Next**, then click **Create File**. Name the file `arty` and click **OK**, then click **Finish**.
3. Expand the **constrs_1** item and double-click `arty.xdc` to open the file.
4. Insert the following text into `arty.xdc`:

```
set_property IOSTANDARD LVCMOS33 [get_ports eth_ref_clk]
set_property PACKAGE_PIN G18 [get_ports eth_ref_clk]
```

5. Press `Ctrl + S` to save the file.

Design entry is now complete for this phase of the project. We will now perform the synthesis, implementation, and bitstream generation steps:

1. Under **Flow Navigator**, click **Generate Bitstream**. Click **Yes** in the **No Implementation Results Available** dialog and **OK** in the **Launch Runs** dialog. This process may take several minutes to complete.
2. When the **Bitstream Generation Completed** dialog appears, click **Cancel**.

If there are no errors reported, this completes the first stage of the oscilloscope FPGA development project. Although we have not implemented any logic related to our ADC interface yet, the current design is capable of booting and running a software application.

Follow these steps to create and run the code for the TCP echo server:

1. In Vivado, select **File | Export | Export Hardware....** Select the **Fixed Platform type**, then click **Next**.
2. In the **Output** dialog, select the **Include Bitstream** option and click **Next**.
3. In the **Files** dialog, select the directory `C:/Projects/oscilloscope-software` and click **Next**. Click **Finish** to complete the export.
4. Locate the desktop icon titled **Xilinx Vitis 2020.1** (or look for your version number, if different) and double-click it.
5. Select the `C:\Projects\oscilloscope-software` directory for your workspace and launch Vitis:

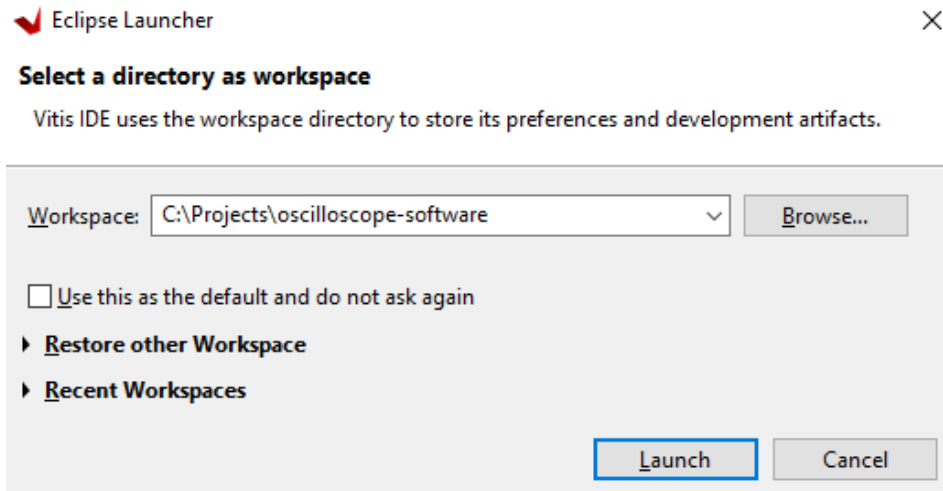


Figure 5.11 – Vitis workspace selection dialog

6. In the Vitis main window, click **Create Application Project**.
7. Click **Next** in the **Create a New Application Project** window.
8. In the **Platform** dialog, click the **Create a new platform from hardware (XSA)** tab.
9. Click the **Browse...** button and locate the hardware definition file. This should be at `C:\Projects\oscilloscope-software\ design_1_wrapper.xsa`. After selecting the file, click **Next** in the **Platform** dialog.
10. On the **Application Project Details** screen, enter `oscilloscope-software` as the **Application project name** and click **Next**.
11. In the **Domain** dialog, select `freertos10_xilinx` in the **Operating System** dropdown. Click **Next**.
12. In the **Templates** dialog, select **FreeRTOS lwIP Echo Server** and click **Finish**.

13. After Vitis finishes setting up the project, it is necessary to change a configuration setting for the Ethernet interface or the application will not work properly. Click **Navigate to BSP Settings** as shown in the following screenshot:

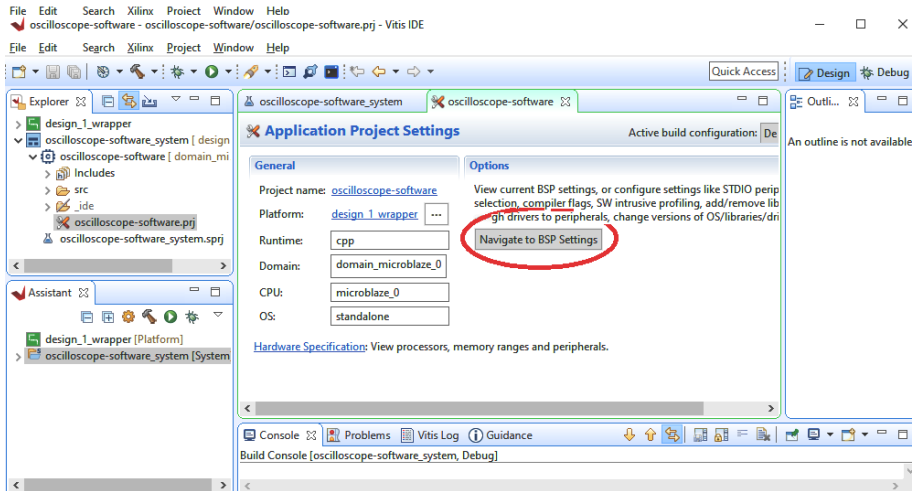


Figure 5.12 – Navigate to BSP Settings button

14. In the **Board Support Package** window, click **Modify BSP Settings...**, then select **lwip211** in the tree on the left. Find the **temac_adapter_options** section in the main window and expand it. Change **phy_link_speed** to 100 Mbps (CONFIG_LINKSPEED100) and click **OK**. This step is necessary because auto-negotiation of link speed does not work properly:

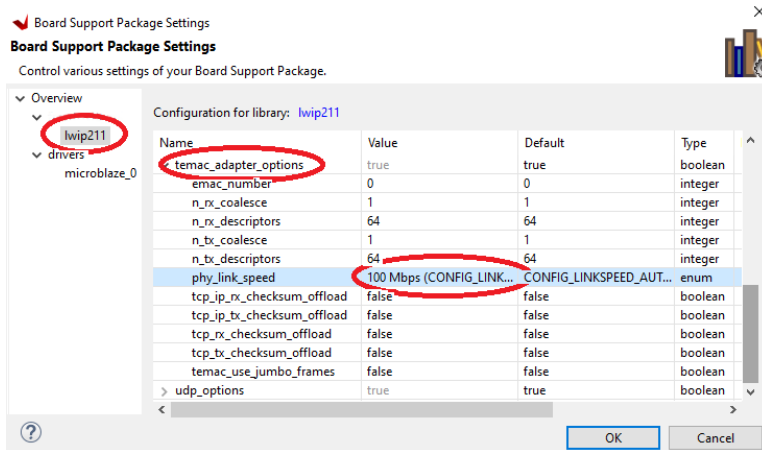


Figure 5.13 – Configuring Ethernet link speed

15. Type **Ctrl + B** to build the project.

If the build process completes with no errors, the result is an executable file that will run on the soft processor in the Arty FPGA. Follow these steps to run the program in the debugger:

1. Connect the Arty A7 board to your PC with a USB cable and use an Ethernet cable to connect the Arty Ethernet port to the switch used by your PC.
2. Use the Windows Device Manager to identify the COM port number of the Arty board. To do this, type `device` into the Windows search box and select **Device Manager**. Expand the **Ports (COM and LPT)** section. Observe the COM port number that disappears and reappears when you disconnect and reconnect the Arty board USB cable.
3. Click the **Debug** icon in the toolbar as shown in the following screenshot. This will load the configuration bitstream and the application you just built into the FPGA:

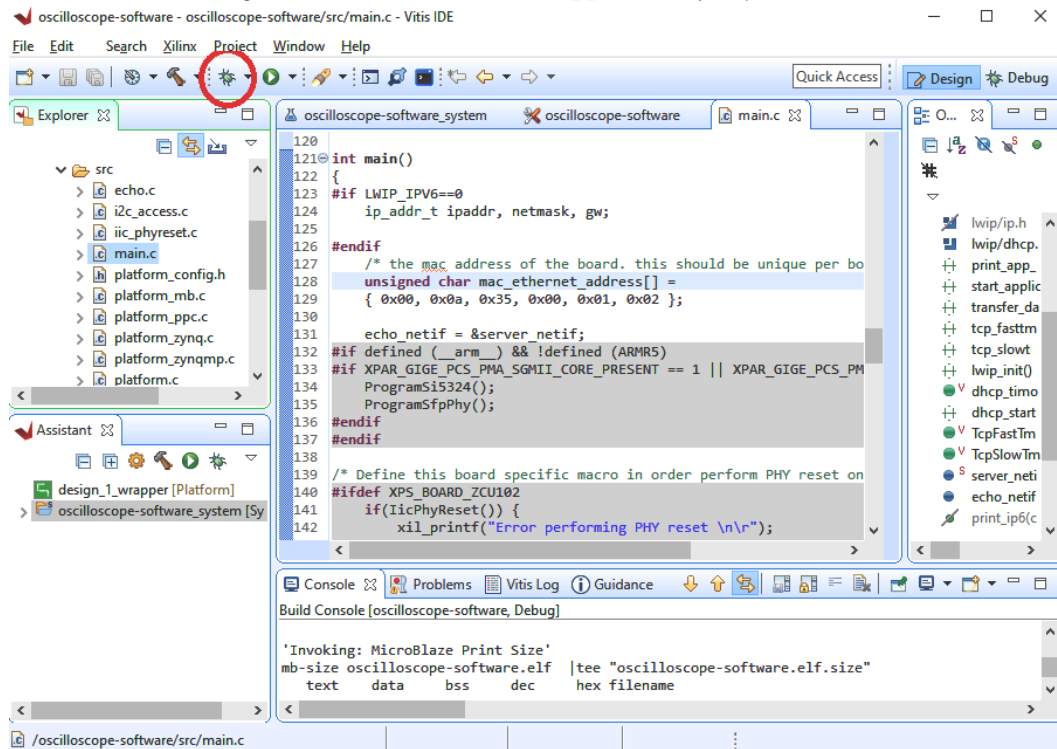


Figure 5.14 – Starting the debugger

4. In the bottom-center dialog area, locate the **Vitis Serial Terminal** tab and click the green + icon to add a serial port:

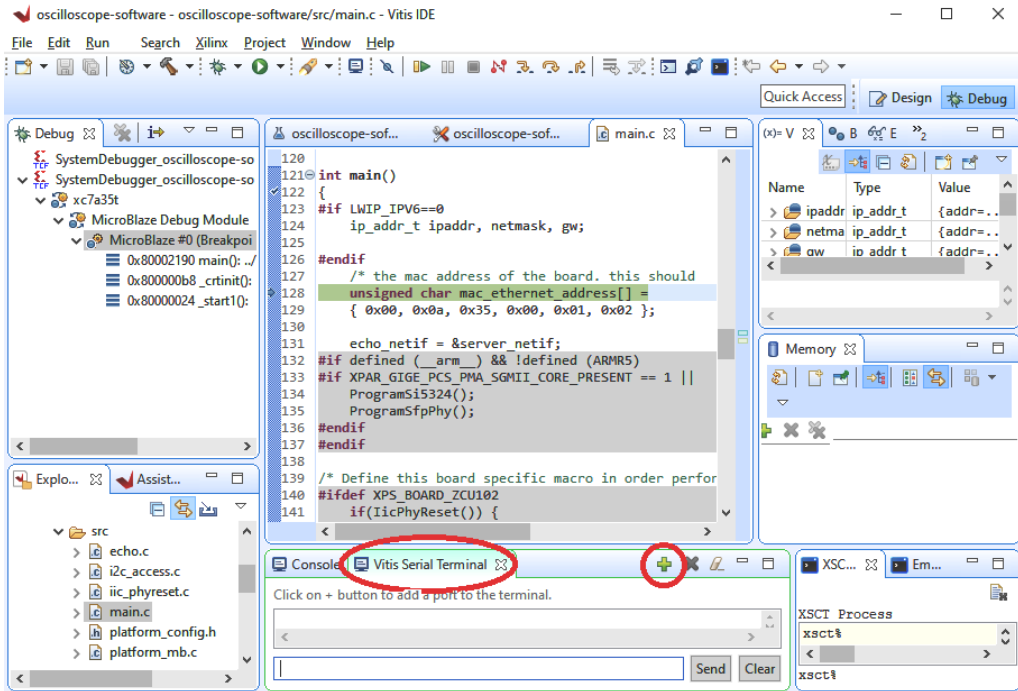


Figure 5.15 – Configuring the serial terminal

5. In the **Connect to serial port** dialog, select the COM port number you identified in the Windows Device Manager and set **Baud Rate** to 9600. Click **OK**.
6. Press **F8** to start the application. You should see output in the serial terminal window similar to this:

```
-----lwIP Socket Mode Echo server Demo Application -----
```

```
link speed: 100
```

```
DHCP request success
```

```
Board IP: 192.168.1.188
```

```
Netmask : 255.255.255.0
```

```
Gateway : 192.168.1.1
```

```
echo server      7 $ telnet <board_ip> 7
```

7. If you do not have `telnet` enabled on your Windows computer, run **Command Prompt** as administrator and enter this command:

```
dism /online /Enable-Feature /FeatureName:TelnetClient
```

8. After the `dism` command completes, close the Administrator Command Prompt and open a user-level Command Prompt.
9. Using the board IP address displayed in the serial terminal window (192.168.1.188 in this example), run `telnet` with this command:

```
telnet 192.168.1.188 7
```

10. Begin typing text in the `telnet` window. For example, typing `abcdefgh` produces the following output, which contains an echo of each character except the first:

```
abbccddeeefgghh
```

11. To verify the echoed characters are coming from the Arty board, break the application execution by clicking the Suspend button as shown in the following screenshot:

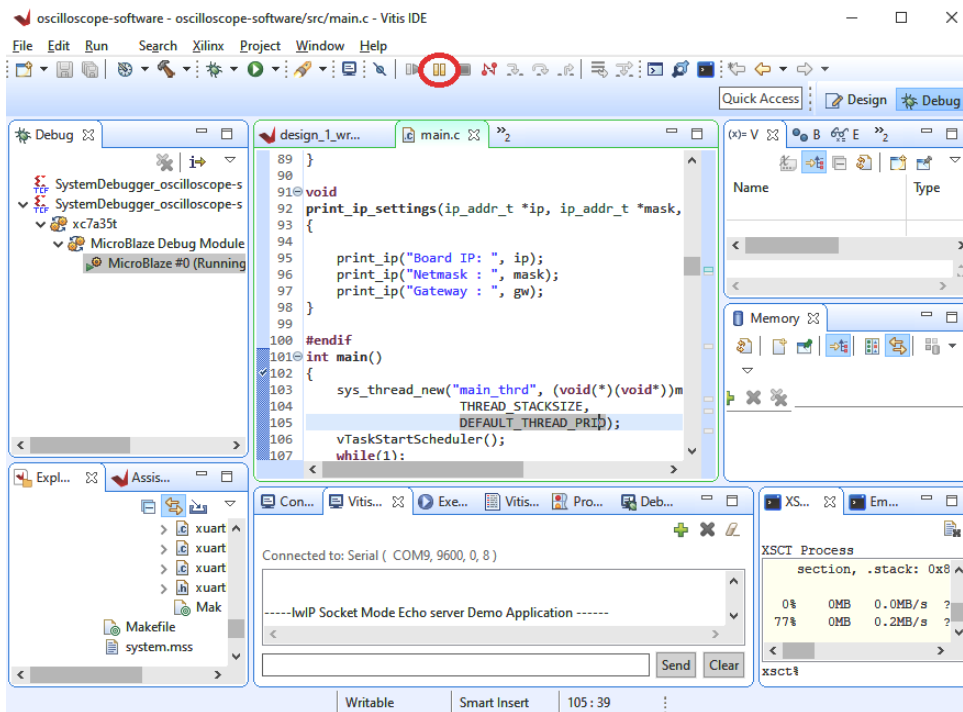


Figure 5.16 – Suspending debugger execution

12. Type some more characters into the `telnet` window. Observe that they are no longer being echoed (only one character appears for each keypress).

This completes the initial implementation phase of the FPGA design and the software application that will run on it. In this chapter, we have developed a baseline Arty A7 MicroBlaze processor system running the FreeRTOS real-time operating system that includes TCP/IP networking.

Summary

This chapter described the process of designing and implementing systems using FPGAs. It began with a description of the FPGA compilation software tools that convert a description of a logic circuit in an HDL into an executable FPGA configuration. The types of algorithms best suited to FPGAs were identified and a decision approach was proposed for determining whether a particular embedded system algorithm is better suited for execution as code on a traditional processor or in a custom FPGA. The chapter concluded with the development of a complete FPGA-based computer system with TCP/IP networking. This project will be further refined into a high-performance network-connected digital oscilloscope in later chapters.

Having completed this chapter, you learned about the processing performed by FPGA compilation software tools. You understand the types of algorithms most suited to FPGA implementation and how to determine whether FPGA implementation is right for a given application. You have also worked through an FPGA design example that will support a high-performance, real-time application in the coming chapters.

The next chapter introduces the excellent, open source KiCad electronics design and automation suite and describes how it can be used to develop high-performance digital circuits. The chapter will continue the example oscilloscope project we began in this chapter, transitioning into the circuit board development phase.

6

Designing Circuits with KiCad

This chapter introduces the excellent, open source KiCad electronics design and automation suite. Working in KiCad, you design a circuit using schematic diagrams and develop a corresponding printed circuit board layout. You'll learn how to turn a circuit board design into a prototype at a very reasonable cost. This chapter includes example schematics for the oscilloscope circuit project you will assemble in the next chapter.

After completing this chapter, you will have downloaded and installed KiCad, learned how to create circuit schematics in KiCad, learned how to develop circuit board layouts in KiCad, and worked through portions of the circuit board design for the digital oscilloscope project.

We will cover the following topics in this chapter:

- Introducing KiCad
- Basic KiCad procedures
- Developing the project schematic diagram
- Laying out the **Printed Circuit Board (PCB)**
- Prototyping the circuit board

Technical requirements

Files for this chapter are available at <https://github.com/PacktPublishing/Architecting-High-Performance-Embedded-Systems>.

KiCad is available to download for free at <https://kicad-pcb.org/download/>. Several operating systems are supported, so be sure to select the correct distribution when you begin the download. When the download has completed, run the installer and accept the defaults at the prompts.

Introducing KiCad

The KiCad suite installs a set of applications that perform the following functions:

- **Schematic entry:** Schematic entry is the process of describing an electrical circuit using a diagram that shows the circuit components and the connections between them. In KiCad, you select from a palette-based set of tools to choose electrical components, arrange them on the drawing canvas, and connect them together with lines that represent wires.
- **Component definition:** KiCad includes a large set of definitions for common electrical components. Additional libraries are available for free from a variety of online sources. Despite the availability of a large set of predefined devices, you will occasionally need to define a component that isn't in the library. KiCad provides tools to describe the electrical connections of a device in terms of pins and their functions, and to define a footprint for the device. The **footprint** of an electrical component describes the connections required to install the component in a circuit board, such as the number and location of holes required and the size and location of metal solder pads.
- **PCB development:** The circuit defined during schematic entry identifies the components and connections required by the circuit. To implement the design in a circuit board, it is necessary to specify the physical location of each component and lay out the connections between the components. These connections, which function as wires between components, are called **traces**.

- **Generation of PCB manufacturing files:** Having defined all aspects of a PCB in KiCad, the next step is to produce a file or set of files that can be used by a PCB manufacturer to produce a circuit board for the design. A standard called the **Gerber format** is used in the PCB industry to specify PCB designs. KiCad provides the ability to generate output in Gerber format for use in PCB manufacturing. Some PCB manufacturers support the KiCad file format directly as their input for PCB production, saving the developer the step of generating Gerber format output.

KiCad supports the design of PCBs with multiple layers. Each layer in a PCB contains a metal sheet that can be selectively removed during manufacturing, enabling the creation of arbitrary wiring connections across the two-dimensional surface. Connections between layers, called **vias**, allow traces to cross each other and enable complex interconnections among densely placed components.

KiCad supports PCBs that use both **through-hole technology (THT)** and **surface-mount technology (SMT)**. As the name indicates, THT devices connect to a PCB by inserting wires or metal pins through holes in the PCB. Solder is then applied to create a strong physical and electrical connection between each device pin and the PCB.

SMT devices, on the other hand, do not require holes in the PCB. Each SMT device is soldered to a set of metal pads on the PCB surface to create the mechanical and electrical connections between the board and the device.

Figure 6.1 shows an example of an SMT device in the lower left and a THT device in the upper right:

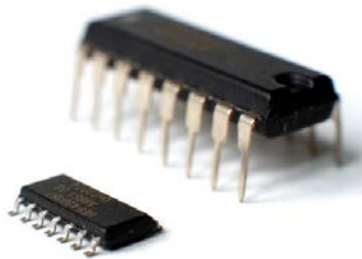


Figure 6.1 – SMT and THT devices

THT device technology for integrated circuits is older and is used less frequently in modern circuit designs. PCBs produced using SMT devices are generally smaller than circuits constructed with corresponding THT devices.

While the integrated circuits we will be working with will generally be of the SMT type, other components on a PCB, such as connectors to external devices or for power supplies, frequently rely on THT components.

Less complex circuit components, such as resistors and capacitors, are available in both THT and SMT packages. As with integrated circuits, SMT resistors and capacitors are generally smaller than the corresponding THT devices. These SMT devices are available in a number of physical package sizes, as shown in *Figure 6.2*:

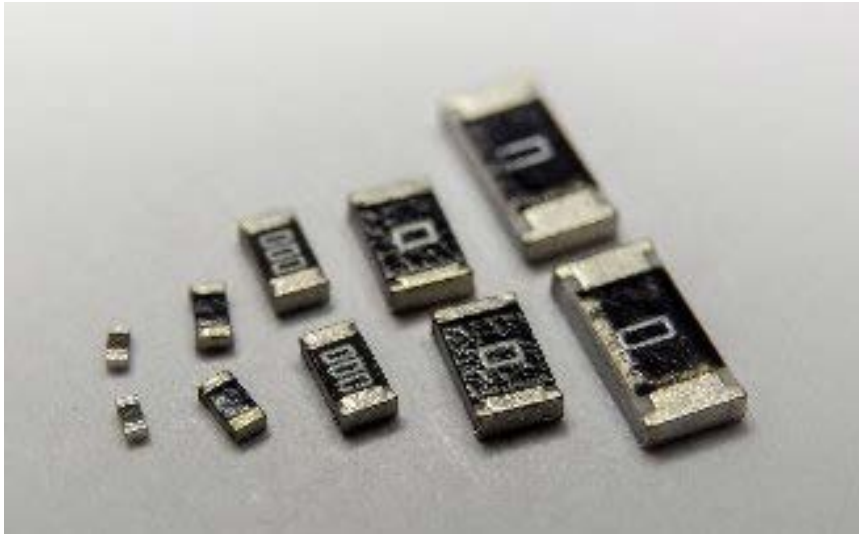


Figure 6.2 – Various SMT resistor packages

Due to the high-performance nature of the circuit we are designing, we will be using some of the more advanced features of KiCad. Specifically, the ADC in our circuit will be generating output data on a set of high-speed differential wire pairs. To maintain the integrity of these signals as they travel through the PCB traces, it is necessary to use the proper PCB layout techniques which we will discuss later in this chapter.

In the next section, we will step through the process of creating a circuit design, creating component definitions, and connecting the circuit elements in a diagram.

Basic KiCad procedures

After installing KiCad, you will find a KiCad icon on your Windows desktop. Double-click the KiCad icon to start the KiCad project management window. This window will appear as shown in *Figure 6.3*:

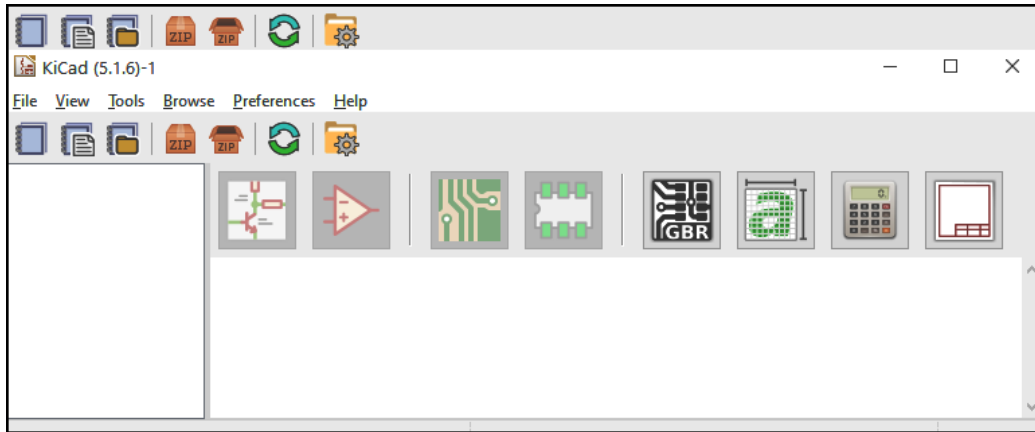


Figure 6.3 – KiCad project manager window

On the **File** menu, select **New | Project...** to create a project. You will be prompted to select a filename and directory location for the project. To begin designing the circuit board for the oscilloscope project we began in *Chapter 5, Implementing Systems with FPGAs*, select the `C:\Projects\oscilloscope-circuit` directory and enter `oscilloscope` as the file name. This will create the schematic and PCB files as shown in *Figure 6.4*:

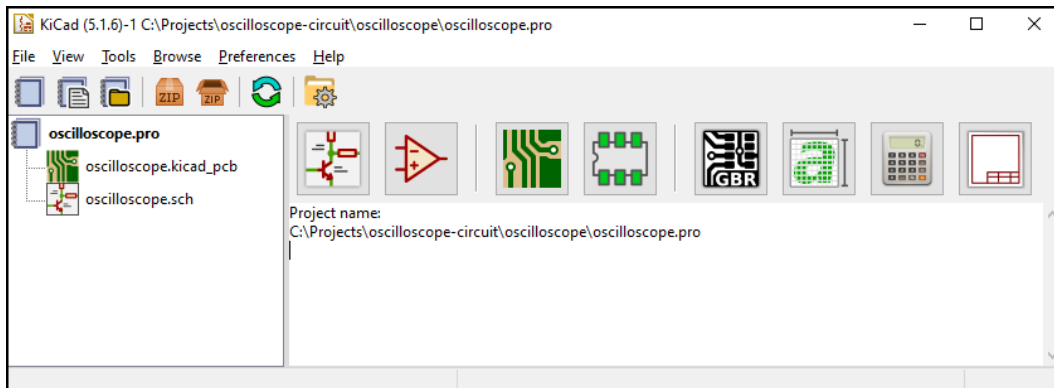


Figure 6.4 – Oscilloscope KiCad project files

As the names suggest, the `.pcb` file name extension contains the PCB description and the `.sch` file name extension contains the schematic.

In the next section, we will begin developing a schematic diagram for two of the oscilloscope power supply voltages.

Placing and connecting circuit components

Double-click the `oscilloscope.sch` icon to open the schematic editor. Before jumping into our circuit design, we will first take advantage of a capability KiCad provides for organizing larger projects: hierarchical sheets.

While the default drawing area provides enough space for a reasonably complex circuit, it is generally a good idea to organize a larger circuit design into a set of drawings with clearly indicated connections between them. We will use hierarchical sheets in KiCad to do this in the steps that follow.

A good place to start a circuit design is with the power supply. Our circuit will connect to the Arty A7-100T board through the rectangular **peripheral module (Pmod)** connectors along the board edge. Each of these connectors provides +3.3VDC and ground connections. The circuits we use will require a few additional power voltages, which means we need to provide regulated power supplies that output each of those voltages.

Perform the following steps to create the power supply schematic as a separate sheet:

1. In the KiCad schematic editor (which is named `Eeschema`), on the **Place** menu, select **Hierarchical Sheet....** Move the cursor to the location where you want to place the sheet and left-click the mouse, then move the mouse and left-click again to finish drawing a rectangle.
2. A dialog will appear prompting for a filename and a sheet name. Enter `Power Supply.sch` as the filename and `Power Supply` as the sheet name.
3. Press the *Esc* key to exit the hierarchical sheet creation mode. You can also click the arrow icon at the top of the tool strip on the right side of the main window to exit any of the modes.
4. Double-click the rectangle you just created to open the **Power Supply** sheet.
5. We'll begin by creating a ground symbol. Click the ground icon as shown in the following figure, then click in the drawing area of `Eeschema`:

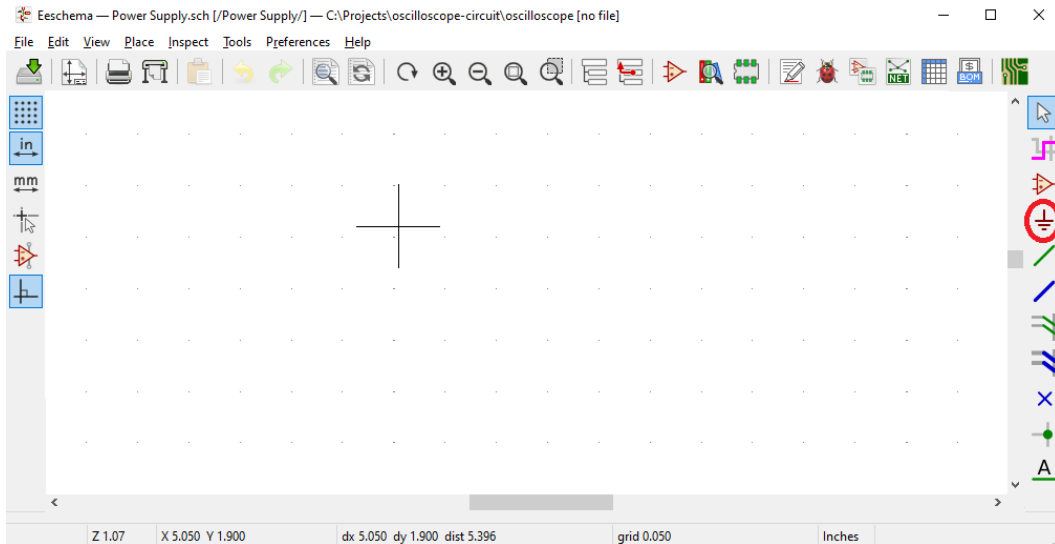


Figure 6.5 – Ground icon in the KiCad toolbar

6. The library of power symbols will load, which may take a few seconds. After the loading completes, click the + next to **power** to expand the list of power symbols. Scroll down to find the symbol named **GND** and click to select it. Click **OK**, then click in the drawing area to place the symbol.
7. For now, we only want to place a single symbol. Press *Esc* to exit symbol placement mode.
8. Repeat the process of clicking the ground symbol icon and clicking in the drawing area, but this time scroll the symbol list to select the symbol named **+3.3V**. Add this symbol to the drawing and press *Esc* to exit placement mode.

9. To define +3.3V as a power net, which means we can connect to this power source anywhere in the schematic by just using the +3.3V symbol, we need to add a flag to each of these symbols. This flag is simply a KiCad component (and not a real circuit component) that tells KiCad to make the power connections associated with these symbols globally available across the circuit. To begin this procedure, click the **Place symbol** icon as shown in the following figure, then click in the drawing area:

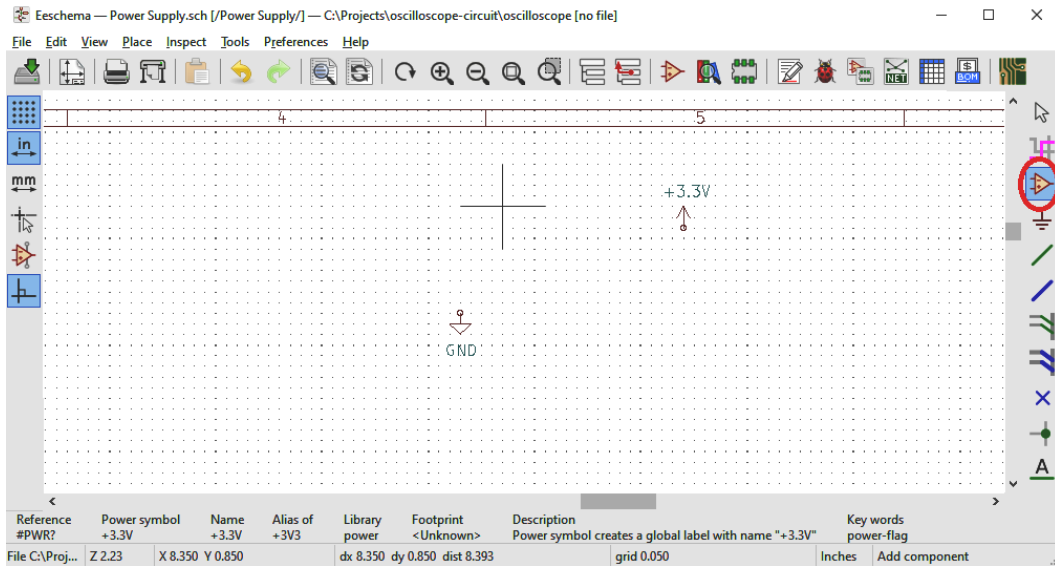


Figure 6.6 – Place symbol tool in the KiCad toolbar

10. The **Choose Symbol** dialog will appear. In the **Filter** field, type pwr. The **PWR_FLAG** symbol will appear in the list. Click to select this symbol, then click **OK**. Click in the drawing area twice to place the component two times. Press *Esc* to exit placement mode.

11. At this point, your diagram should look something like this:

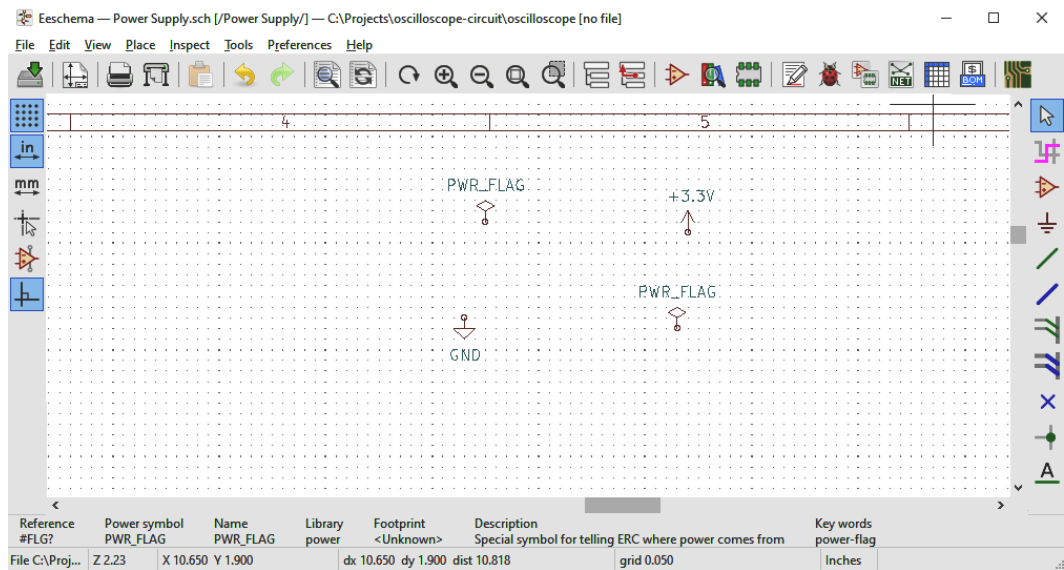


Figure 6.7 – Power, ground, and power flag symbols

12. One way to move one or more symbols on the diagram is to select them by dragging a box over them, then move the mouse to reposition them, and finally left-click to set the new position. Do this to place the **GND** symbol directly beneath a **PWR_FLAG** symbol and place the other **PWR_FLAG** symbol beneath the **+3.3V** symbol.

13. You can rotate a symbol 90° by hovering the cursor over the symbol and pressing the **R** key. Do this two times on the **PWR_FLAG** symbol beneath the **+3.3V** symbol. Move the symbols to get everything nicely aligned as in the following figure:

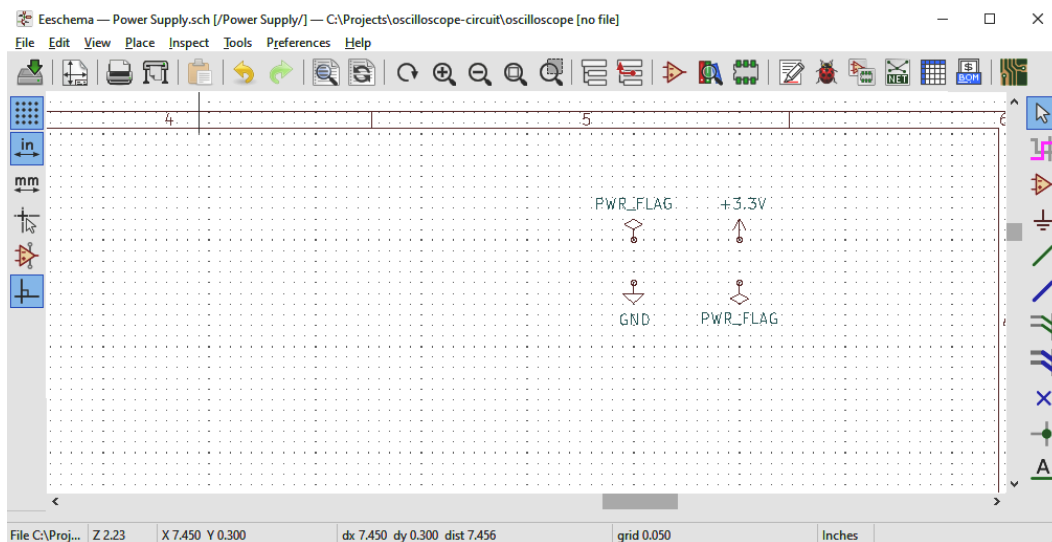


Figure 6.8 – Symbols after repositioning and rotating

14. Next, we will draw connecting wires. Click the *Place wire* icon as shown in the following figure, and then click at the start and end of each wire connection to complete the connections:

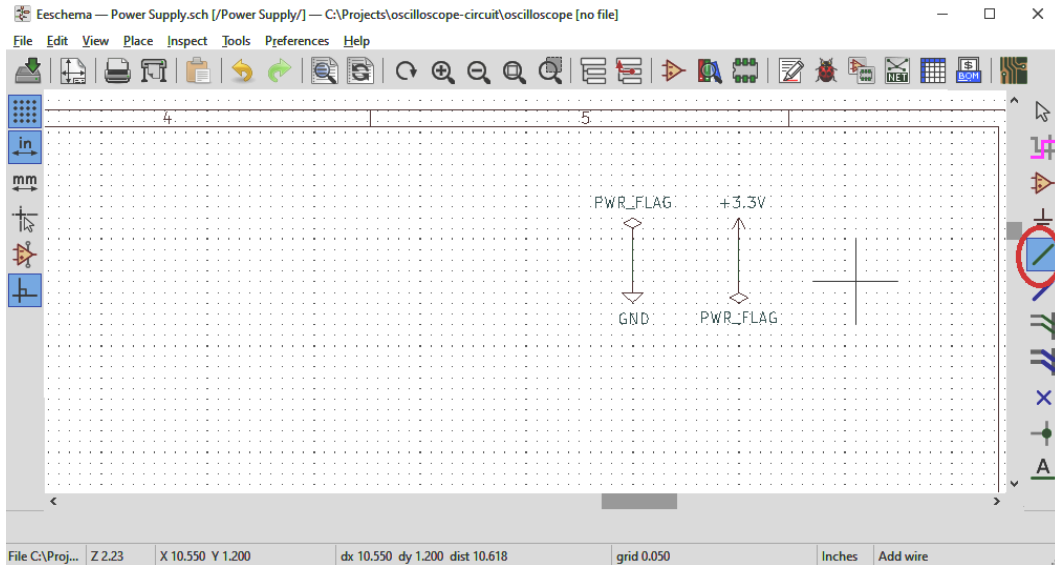


Figure 6.9 – Connections wired between components

We will now add a voltage regulator to the circuit to provide a +1.8 VDC supply for the ADC. The Texas Instruments TLV757P is a 5-pin integrated circuit capable of producing a regulated 1.8 V output from a 3.3 V input voltage.

When developing a schematic that includes any type of integrated circuit, it is important to acquire the device data sheet and become familiar with the contents. Data sheets for integrated circuits contain a wealth of information describing the device capabilities, operational limits, and recommendations for successful implementation. The data sheet for the TLV757P is available at <https://www.ti.com/lit/ds/symlink/tlv757p.pdf>.

From a review of the data sheet, we see that the TLV757P requires two 1 μ F capacitors, one connected between the source voltage and ground and the second between the output voltage and ground.

Add the +1.8 VDC power supply to the schematic with the following steps:

1. Click the *Place symbol* icon as shown in *Figure 6.6*, then click in the drawing area where you would like to place the symbol. The symbol libraries will load.
2. In the **Choose Symbol** dialog's **Filter** field, type `tlv757` and observe the list of devices that appears.
3. The listed components are variations of the TLV757P device with different fixed output voltages and package types. Search through the list to find the part with an output voltage of +1.8 V and with a package type of SOT-23-5. Select the entry and click **OK**.
4. Click in the drawing area where you would like to place the device, then press *Esc* to exit placement mode.
5. We will next add two unpolarized capacitors to the diagram. **Unpolarized capacitors** do not have positive or negative terminals. **Polarized capacitors**, in comparison, must have the positive terminal connected to the higher voltage of the two terminals. Click **Place symbol** again, click in the drawing area, and this time type `capacitor` in the **Filter** field.
6. Scroll down the list to find a capacitor with the device name C. Place two copies of the capacitor on the drawing, one to the left of the TLV757P and one to the right.
7. We need to label the capacitors to indicate their capacitance. Double-click the text for the C label next to one of the capacitors (don't click the C? label) and enter the text `C_1u`. Repeat for the second capacitor.
8. Connect pin 2 of the TLV757P to **GND**. Do not draw a line to the ground symbol we added earlier. Instead, add a new **GND** symbol beneath the TLV757P and connect to that. Connect the lower terminals of both capacitors to **GND**.
9. Click the ground icon as shown in *Figure 6.5*, then click in the drawing area and type `1.8` in the **Filter** field. Select the symbol named **+1V8**, click **OK**, and place the symbol above the capacitor at the right of the TLV757P. Draw a wire to connect this symbol to pin 5 of the TLV757P.

Important note

Power supply voltages in schematic diagrams: The letter V is sometimes used in place of a decimal point when specifying supply voltages in electrical schematic diagrams. The **+1V8** symbol uses this convention.

Finish connecting the TLV757P pins and capacitor terminals as shown in the following figure:

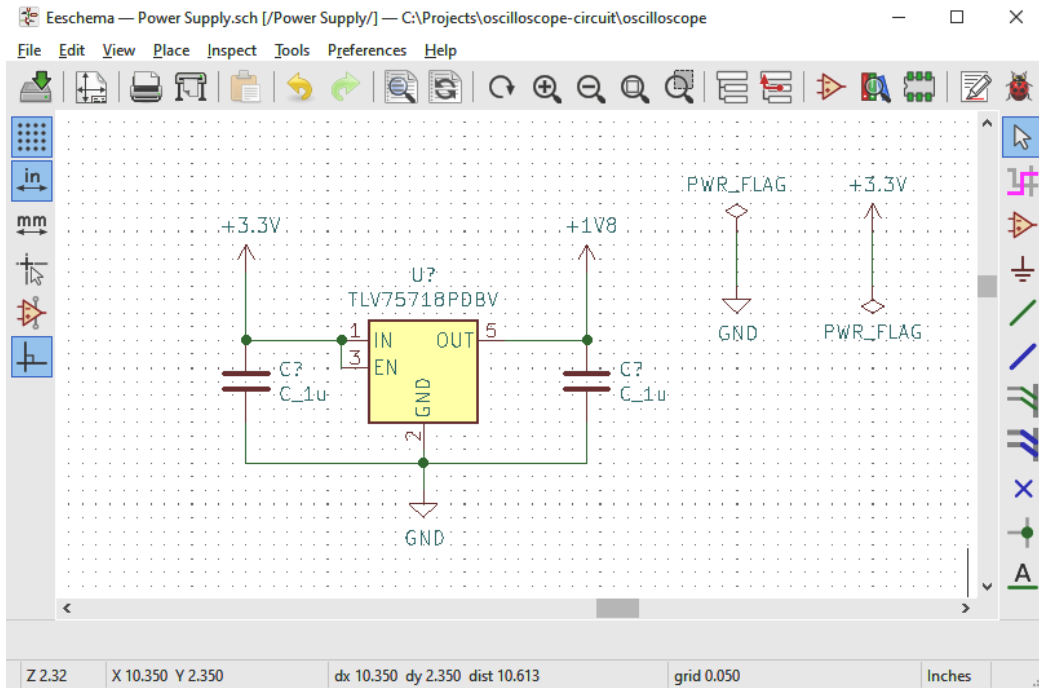


Figure 6.10 – +1.8 V power supply

The following notes will help you navigate within KiCad and become more proficient in your use of the KiCad schematic editor:

- You can leave a hierarchical sheet and return to the higher-level sheet by right-clicking and selecting **Leave Sheet** from the context menu.
- KiCad supports a number of shortcuts for common operations. For example, you can create a copy of a single component or a group of components by holding down the *Shift* key while dragging a box that touches all the components of interest. After releasing the mouse button, move the group to the location where you want to place them and left-click. To see a list of all hotkeys, press *Ctrl + F1*.

- You can quickly zoom in, zoom out, and reposition the drawing in the window using the mouse wheel. Place the mouse cursor over the drawing location you would like to become centered, then roll the wheel forward to zoom in and center, or roll backward to zoom out and center.
- If you drag a rectangle to select one or more components (without holding the *Shift* key down), the default operation is to disconnect the wires from those components and move only the components. If you press the *Tab* key after selecting the components, the connections to the components will remain in place during the move. Try this with the power supply schematic you just created.

At this point, you know much of what is required to draw a schematic diagram in KiCad. Drawing a circuit consists largely of selecting components, placing them on the drawing surface, drawing connecting wires, and rearranging the drawing elements for clarity as the schematic becomes more complex.

When creating schematic diagrams, it is worthwhile to take a little time to organize the drawing so functional areas are separated, components are not placed too close to one another, and wire connections are easy to follow. Wires should never cross other wires or components unless absolutely necessary.

KiCad contains a large library of predefined components you can use directly in your schematic diagrams. Unfortunately, the library does not contain all possible circuit components, so at times you will find it necessary to create a symbol for a component. The next section leads you through the process of creating schematic symbols for circuit components.

Creating component symbols

Our circuit requires **Electrostatic Discharge (ESD)** protection on the signal input to avoid damaging circuit components when, for example, a user walks across synthetic carpet and touches the signal input.

A component called a **Gas Discharge Tube (GDT)** provides the protection we require. A GDT normally behaves as an open circuit, but when the voltage across the GDT exceeds a threshold (75 V for the component we will use) the gas within the device becomes conductive and shorts the electrical energy to ground, protecting sensitive circuit components.

We will use a Littelfuse CG7 Series GDT for this project. This device does not appear in the KiCad library, so we will construct a schematic symbol for it. The symbol is used in schematic diagrams. The Littelfuse CG7 Series data sheet is available at https://www.littelfuse.com/~media/electronics/datasheets/gas_discharge_tubes/littelfuse_gdt_cg7_datasheet.pdf. This document contains all the device-specific information we will need to construct the schematic symbol.

Click the **Symbol Editor** in the top toolbar as indicated in the following figure to start the process:

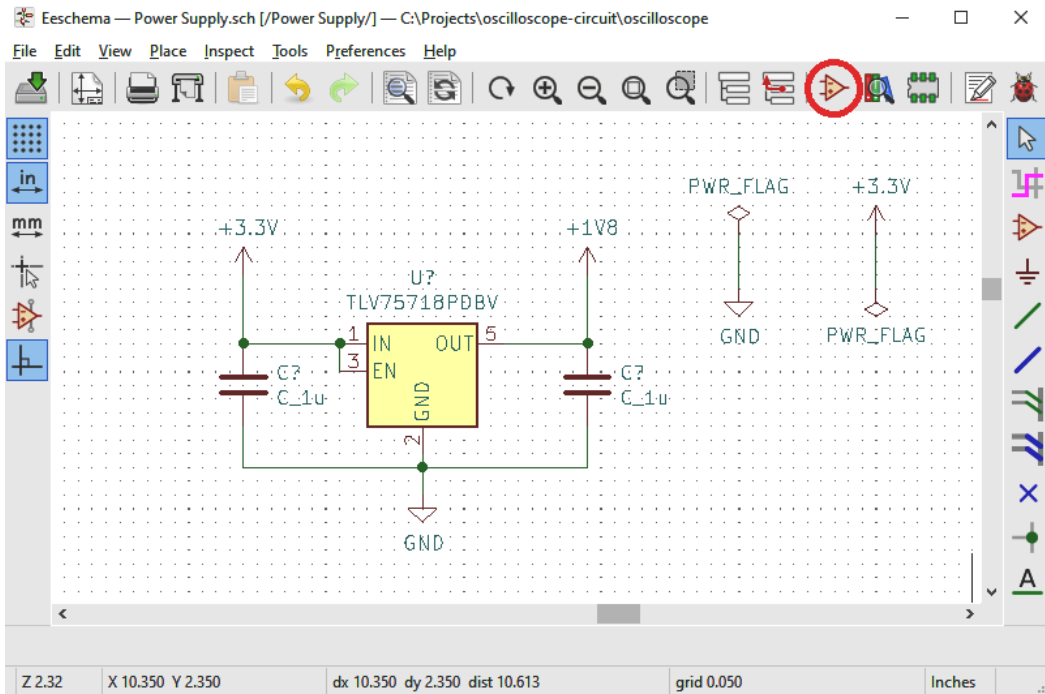


Figure 6.11 – Starting the Symbol Editor

Upon entering the **Symbol Editor**, you will be presented with a list of available symbols.

Perform the following steps to create a symbol for the CG7 GDT:

1. Before you can begin creating symbols, you must first create a library to contain the m. It is a good idea to store your symbol library at an appropriate location for use in future projects. We will create a new library in the `C:\Projects\kicad-symbol-library` directory.
2. In the **Symbol Editor** | **File** menu, select **New Library...** Create the library in `C:\Projects\kicad-symbol-library` with the name `my-symbols.lib`.
3. A prompt will appear asking you to choose between a **Global** or a **Project Library** table. Select **Global** and click **OK**. This choice allows you to use the symbols you create in future projects.
4. Click the toolbar icon to create a new symbol as shown in the following figure:

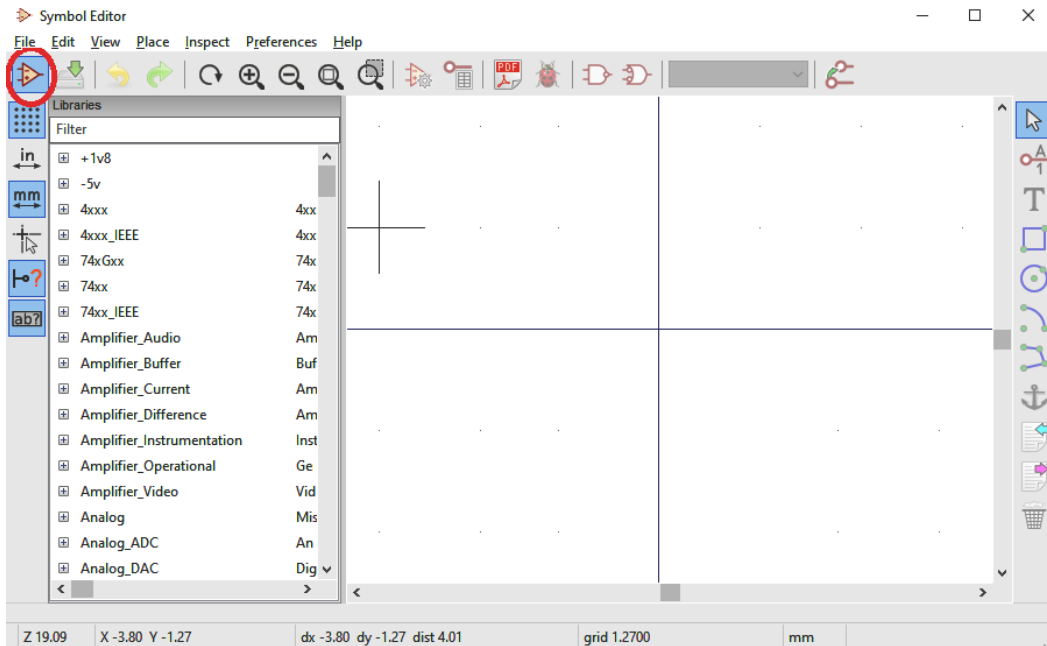


Figure 6.12 – Creating a new symbol

5. You will be prompted for a library to hold the new symbol. Type `my-symbols` in the **Filter** box, select your new library, and click **OK**.
6. A dialog box will appear requesting information about the new symbol's properties. Enter `CG7_GDT` as the symbol name and click **OK**. At this point, the **Symbol Editor** window should appear as shown in the following figure:

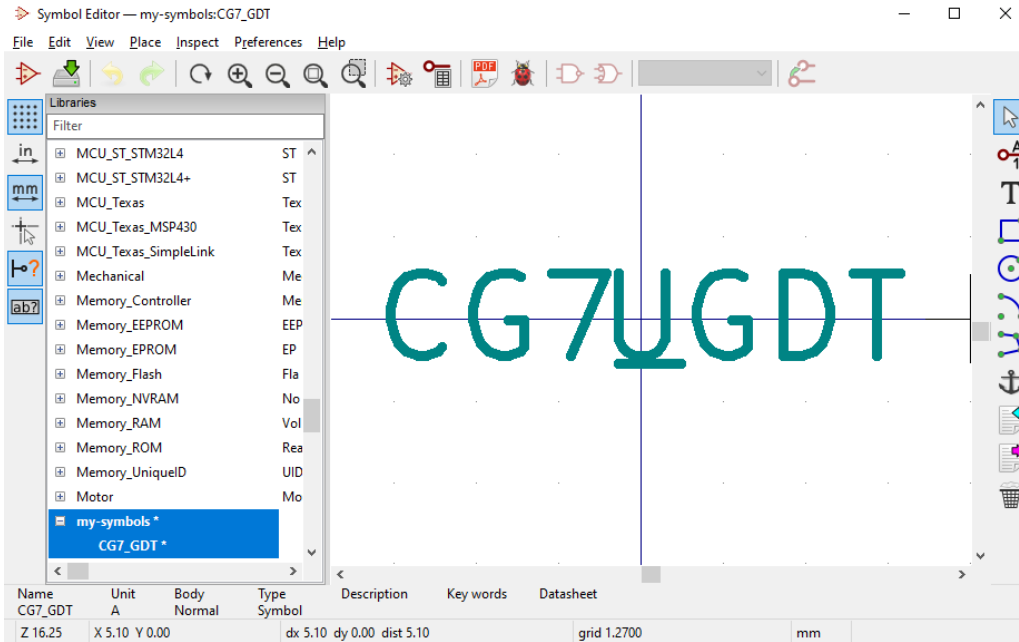


Figure 6.13 – Symbol Editor after beginning the definition of CG7 GDT

7. Select the *Pin* tool (the icon just under the arrow highlighted in the upper-right portion of Figure 6.12) and click in the drawing area.
8. A **Pin Properties** dialog will appear. Enter the following information for the first pin: **Pin name:** ~; **Pin number:** 1; and **Electrical type:** Input. Click **OK**.
9. Carefully move the cursor (while using the mouse wheel to zoom in and out as needed) to position the pin at X -6.35 and Y 0.00 as indicated in the status line at the bottom of **Symbol Editor**. When the proper position has been reached, click the mouse to place the pin.
10. Repeat the procedure for pin 2, also named ~, pin number 2, and with electrical type Input. After clicking **OK** in the **Pin Properties** dialog, press the *R* key two times to rotate the pin through 180° . Place this pin at X 6.35 and Y 0.00 .
11. Click the circle icon in the right-side toolbar, then click at the center of the crosshairs between pin 1 and pin 2. Move the mouse to increase the circle radius, and click the mouse when the circle passes through the small circles for pin 1 and pin 2.
12. We are going to add some graphic shapes to the symbol, but first we need to change the grid size. Right-click on the background of the **Symbol Editor** and select **Grid**, then select a grid size of 10 mils.

13. Add a circle and two triangles to the diagram to replicate the circuit symbol for the CDT as shown in the CG7 Series data sheet. Use the circle and polygon tools in the right toolbar to draw shapes similar to the GDT symbol shown in the data sheet.
14. After drawing each circle and triangle, right-click on the shape and edit the options for the shape. Select **Fill with body outline color** and click **OK**.
15. Move the symbol name (**CG7_GDT**) and reference indicator (**U**) to more suitable locations by right-clicking each text field and selecting the option to move the field. The completed symbol should appear as shown in the following figure:

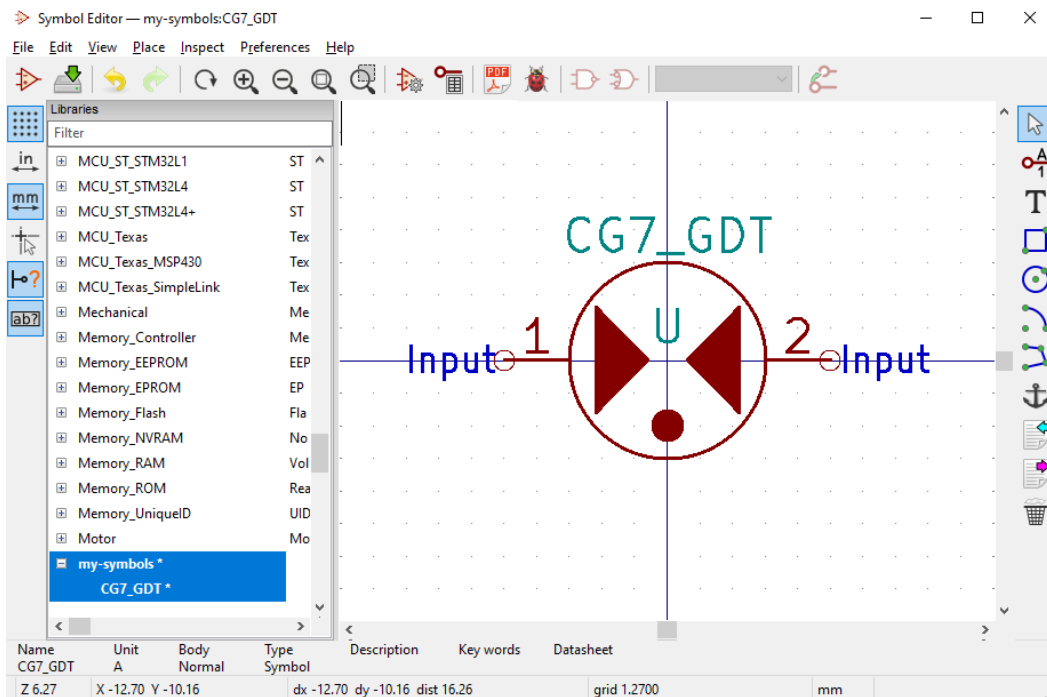


Figure 6.14 – Completed GDT schematic symbol

16. Select **Save All** on the **File** menu, then exit **Symbol Editor**.
17. Return to the **Eeschema** editor window. On the **Preferences** menu, select **Manage Symbol Libraries....** Scroll to the bottom and verify that **my-symbols** appears under the **Global Libraries** tab. Click **Cancel** to return to the editor window.
18. Select the **Place symbol** icon and click in the drawing window. In the **Filter** box, type GDT. Your new symbol should appear listed under **my-symbols**. Select the symbol and click **OK**. Place the symbol in a location with some space around it.

19. For the time being, we will leave the leads to the GDT unconnected. To avoid warnings during the electrical rules check, we can mark the pins as unconnected. Select the blue X icon in the right-side toolbar and click each of the pins on the GDT. This will place a blue X on each of the pins, marking them as unconnected.

By completing these steps, you have created a new component symbol and added it to a circuit schematic diagram. The skills you've learned in this section provide most of the knowledge needed to complete the schematic diagram for the project circuit. The next section will introduce the remaining KiCad features used in developing the oscilloscope project schematic diagram.

Developing the project schematic diagram

The entire circuit diagram for the oscilloscope project is organized into six hierarchical KiCad sheets as shown in the following figure:

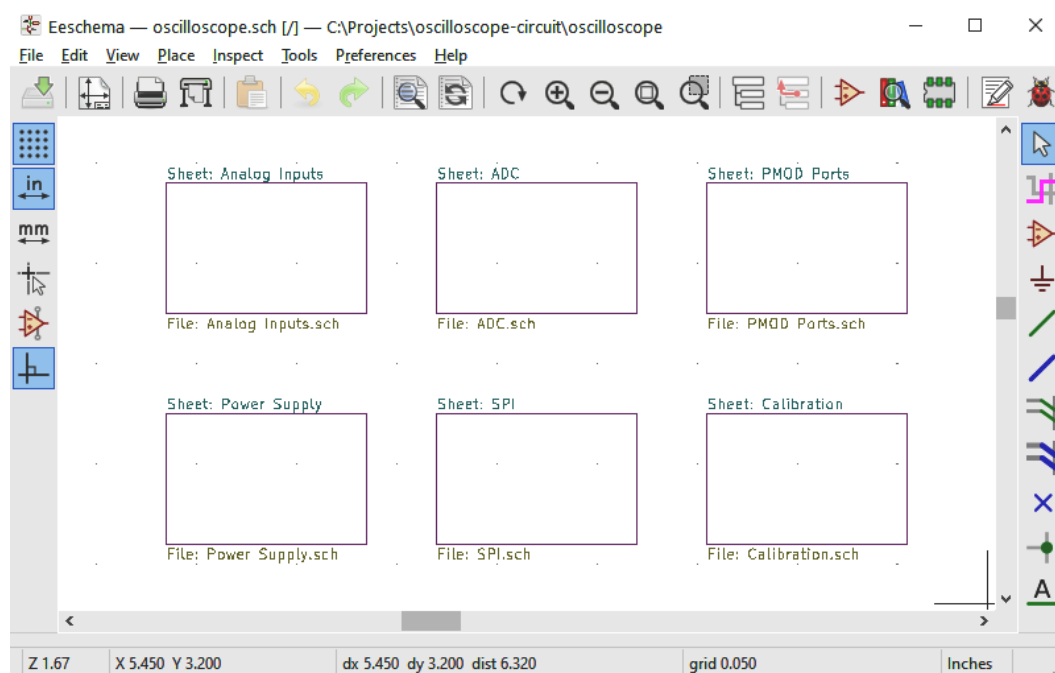


Figure 6.15 – Project schematic sheets

The contents of each sheet are as follows:

- **Analog Inputs:** This portion of the circuitry receives an analog input from a standard oscilloscope probe in the range ± 10 V and transforms it into a differential signal in the range ± 1.0 V for input to the ADC.
- **ADC:** This portion of the circuit connects the ± 1.0 V analog signal to the ADC input pins. This diagram also connects a 100 MHz digital clock signal from the Arty board to the ADC. The ADC provides high-speed LVDS differential outputs for two lanes (*OUT1A* and *OUT1B*) as well as a data clock (*DCO*) that connect to Arty board inputs. The ADC used in the project is the Linear Technology LTC2267-14, a dual-channel 14-bit ADC capable of 105 million samples per second. We will only be using one of the channels in this project. The data sheet is available at <https://www.analog.com/media/en/technical-documentation/data-sheets/22687614fa.pdf>.
- **Pmod Ports:** This sheet describes the connections to the two 2x6 pin connectors that will attach to the Arty board Pmod B and C connectors.
- **Power Supply:** This sheet contains the circuitry for the +1.8 V power supply we developed earlier in this chapter as well as circuits for +2.5 V and -2.5 V supplies that drive the analog circuit components.
- **SPI:** This sheet contains circuit connections for the SPI connector on our circuit board. This interface will connect to the Arty SPI connector using a short ribbon cable.
- **Calibration:** This sheet contains circuitry to provide a 1 KHz output signal alternating between +2.5 V and -2.5 V on a test point. Additional connections are provided for +2.5 VDC and -2.5 VDC test points as well as **GND**.

We will not be going through all six of the diagrams in detail in this chapter, but the next section will describe the additional KiCad features used in developing the diagrams that we haven't already covered. With this information, after downloading the files from the book website, you will be able to trace the flow of signals throughout the circuit and understand how the diagram was constructed.

Adding text annotations

You can add explanatory notes to your diagram such as the text `Input range  $\pm 10$  V` at the left side of the following figure:

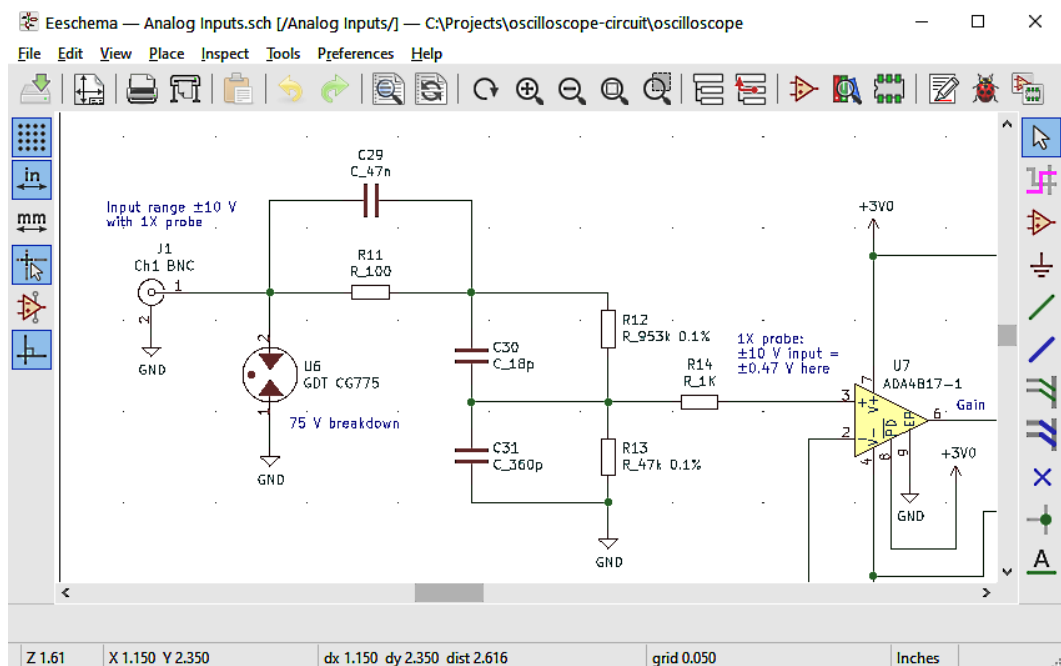


Figure 6.16 – Analog Inputs diagram with text annotations

Enter text annotation mode by clicking the large **T** icon at the bottom of the toolbar along the right edge of the **Eeschema** main window. You'll need to make your window larger than the figures in this chapter to be able to see all of the toolbar icons.

Adding signal labels

You can add labels to signals to indicate their function. Click the **A** icon in the right-side toolbar with the green line below it to enter signal labeling mode. When you click on a signal line, you will be prompted for the signal name.

Adding global labels

The signal labels discussed in the previous paragraph describe connections in a single drawing sheet. You can create connections between sheets with **global labels**. The icon for the global label tool is in the right toolbar, located just beneath the signal label tool. The global label tool resembles an **A** inside a pennant.

In the following figure, the **Ain1+** and **Ain1-** signals at the right side terminate in global labels:

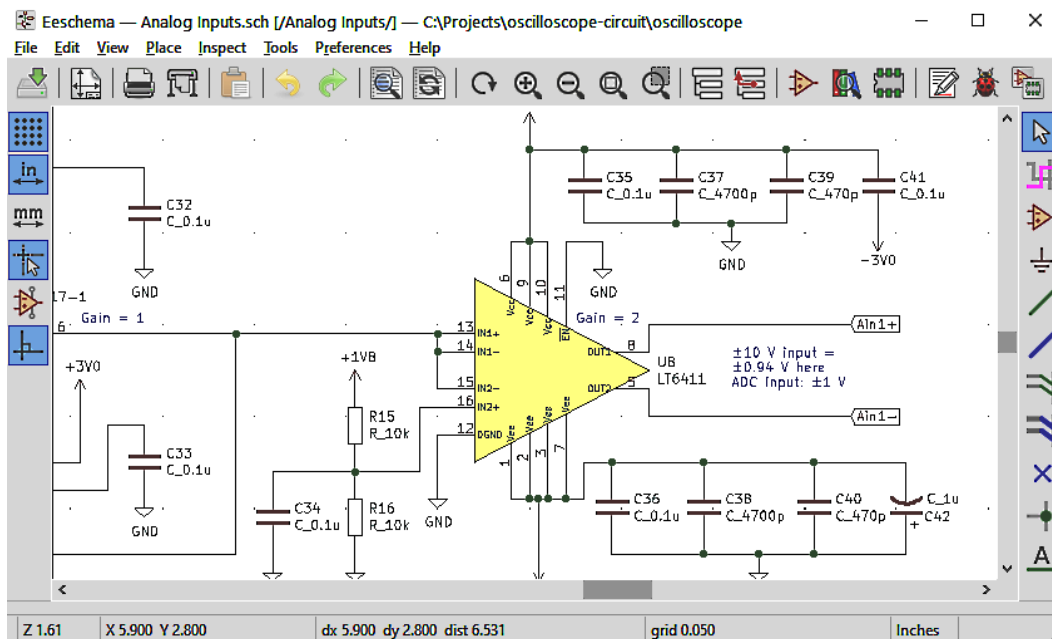


Figure 6.17 – Global labels in the analog Inputs diagram

These signals form the differential pair that provides the analog input to the ADC.

Creating differential signal pairs

KiCad provides robust support for high-speed differential signals in schematic development and in PCB layout. To indicate a differential signal pair in a schematic, it is sufficient to create two signals with identical names except that one of the signal names ends with a + character and the other ends with a - character, as with the **Ain1+** and **Ain1-** signal pair in *Figure 6.17*.

Differential pair signal names can be defined with pin names on a schematic symbol, by using signal labels, or with global labels.

Creating offboard connections

Most circuit designs require interfaces to external components to provide power and transfer data. In KiCad, these connections are specified in the same manner as circuit components. You can search in the libraries for various standard connector types and, if necessary, you can create your own connectors.

The following figure shows the connections to the two 2x6 pin connections to the Arty board:

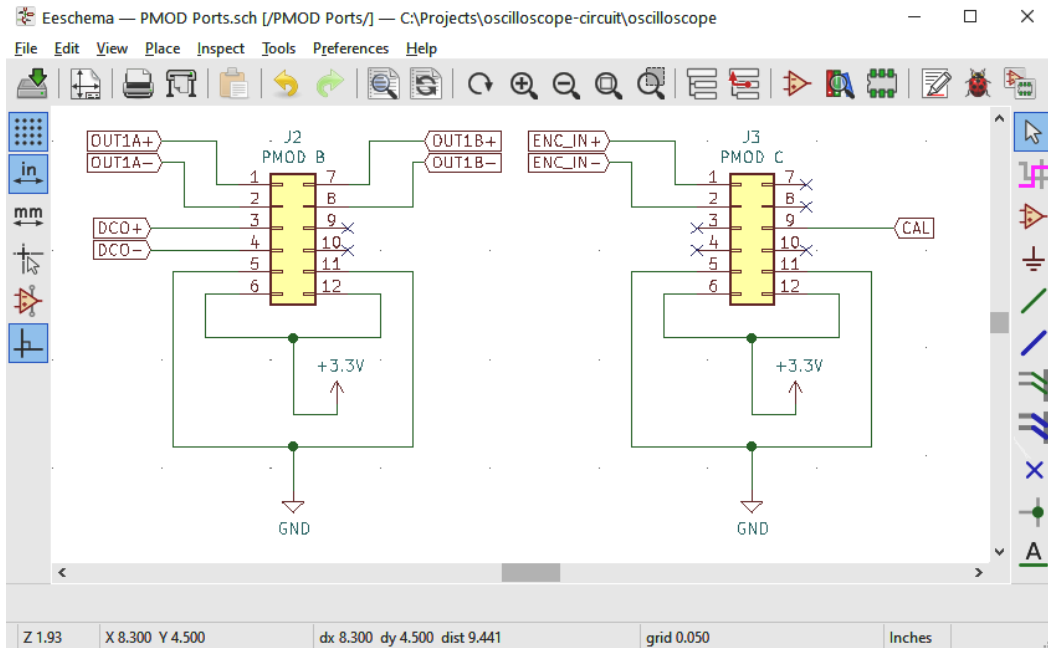


Figure 6.18 – Connectors to the Arty board

The **+3.3 V** and **GND** connections in this diagram are the power source inputs to our circuit board.

Symbol annotation and electrical rules checking

After you have completed drawing a circuit in KiCad, the next step is to annotate the schematic symbols by replacing the question marks in the symbol names with unique numerical values. KiCad will do this automatically with the annotation tool, which is the icon in the top toolbar that resembles a pencil over a sheet of paper. Click the annotation tool icon to bring up the **Annotate Schematic** dialog. Click **Annotate** to assign component numbers, then click **Close**.

Next, you should perform an electrical rules check on the drawing. This is a review of your diagram that identifies potential errors such as unconnected component pins.

To perform the electrical rules check, click the icon on the upper toolbar that resembles a ladybug. The **Electrical Rules Checker** dialog will appear. Click the **Run** button to perform the analysis. If any problems are discovered, you can click the associated link and KiCad will highlight the circuit component that has the issue. Fix any problems before proceeding with PCB layout.

This section provided an introduction to schematic circuit development in KiCad. The next section will show you how to convert the circuit design into a PCB layout suitable for manufacturing.

Laying out the PCB

Once we have a completed, rules-checked schematic diagram, the next step is to begin PCB layout. Before laying out the circuit board itself, we must first assign footprints to each of the circuit components. KiCad maintains schematic symbols and device PCB footprints as separate entities, allowing the user to associate the correct footprint with each device.

We will continue the schematic diagram containing the +1.8V power supply and the GDT we created earlier in this chapter. Click the **Assign PCB footprints to schematic symbols** icon, which is just to the right of the electrical rules check icon. This will open the **Assign Footprints** dialog and list the components in the circuit as shown in the following figure:

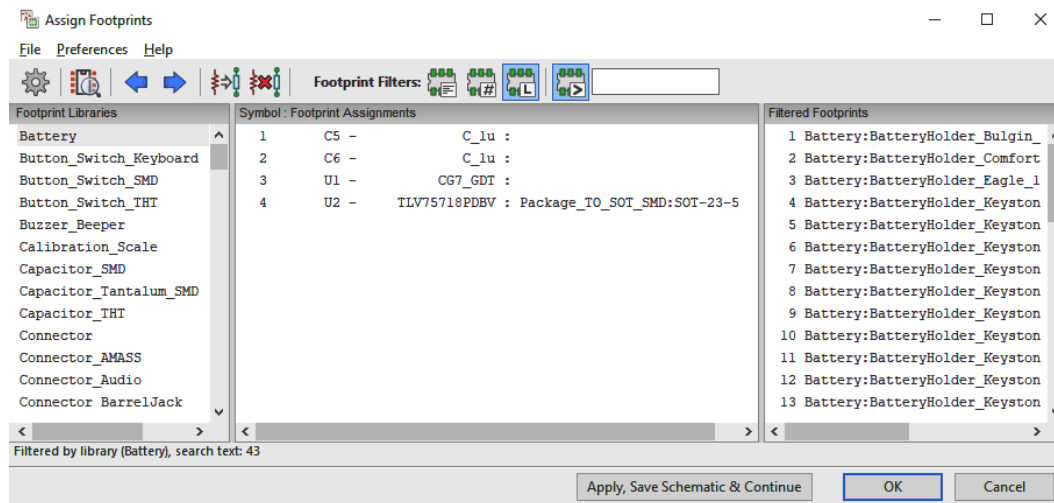


Figure 6.19 – Footprint assignment dialog

Of the four components in our circuit, only one, the TLV757 voltage regulator, already has a footprint assigned. Perform the following steps to assign the remaining components:

1. Click the **Capacitor_SMD** library name in the left column to list the predefined surface mount capacitor footprints in the right column. To assign a footprint, first select a component in the center column, then double-click a footprint in the right column. As you select components in the center window, the corresponding symbol will be highlighted in the **Eeschema** circuit diagram. This helps you keep track of where each component is located in the circuit.
2. We will use the `Capacitor_SMD:C_0402_1005Metric` footprint for the 1 μF capacitors. Click one of the capacitor symbol names in the center-column list. Type `0402` in the search box to filter the footprint list. Assign the footprint to both of the capacitors.
3. The GDT is the only remaining component without a footprint. From the part's data sheet, we see the recommended solder pad sizes are 1.2 x 4.0 mm, separated by 2.5 mm. KiCad provides a footprint editor, enabling the creation of arbitrary solder pad and hole configurations for SMT and THT components. The footprint editor, like other KiCad applications, is straightforward and easy to use. Instead of constructing a footprint from scratch, you can use the footprint editor to modify the pads of an existing footprint with pads similar in size and spacing to the CG7 GDT, such as the `Fuseholder_Littelfuse_Nano2_157x` footprint. Save the modified footprint to your symbol library, then assign the symbol to the GDT in the **Assign Footprints** dialog.
4. Click the **Apply, Save Schematic & Continue** button at the bottom of the dialog, then click **OK**.

We are now ready to begin the layout of the PCB. Open the KiCad PCB editor from the KiCad project manager window. The following figure indicates the icon to click in the project manager:

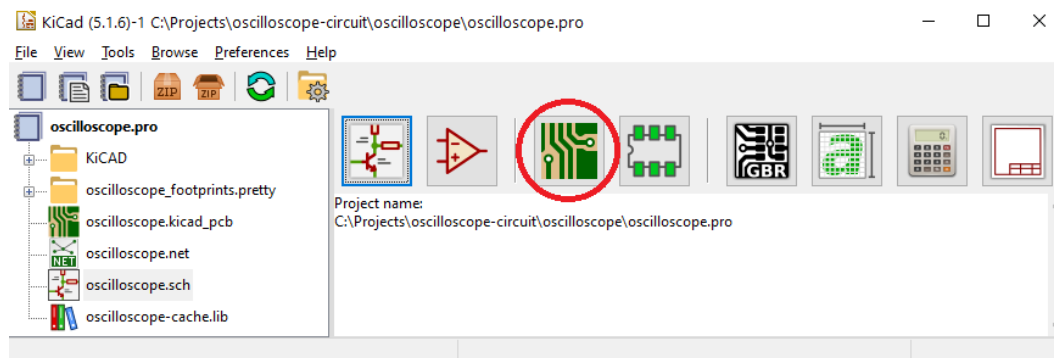


Figure 6.20 – Opening the KiCad PCB layout application

The KiCad PCB layout editor will open with a blank drawing area. Import the circuit component and connection information from the schematic by clicking the **Update PCB from schematic** toolbar icon as shown in the following figure:

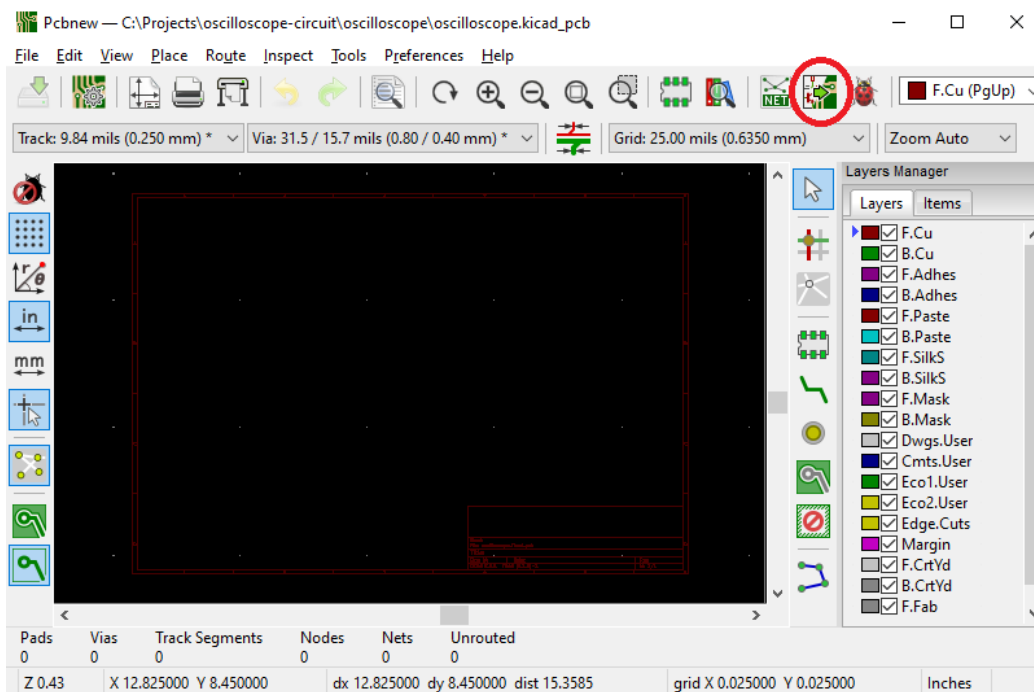


Figure 6.21 – The Update PCB from schematic toolbar icon

Click the **Update PCB** button in the **Update PCB from schematic** dialog, then click **Close**. This will leave you with all of the circuit components in a clump attached to the cursor. Move the cursor to a location to the side of where you plan to draw the circuit board and left-click to drop them there.

Next, we will draw the outline of our circuit board. Set the grid size in the **Grid** dropdown to 100 mils. Select the **Edge.Cuts** PCB layer and select the line drawing tool as shown in the following figure. Click and draw a square 2.5" on each side. Monitor the **Length** and **Angle** indications in the status bar to ensure the lines are straight and of the correct length:

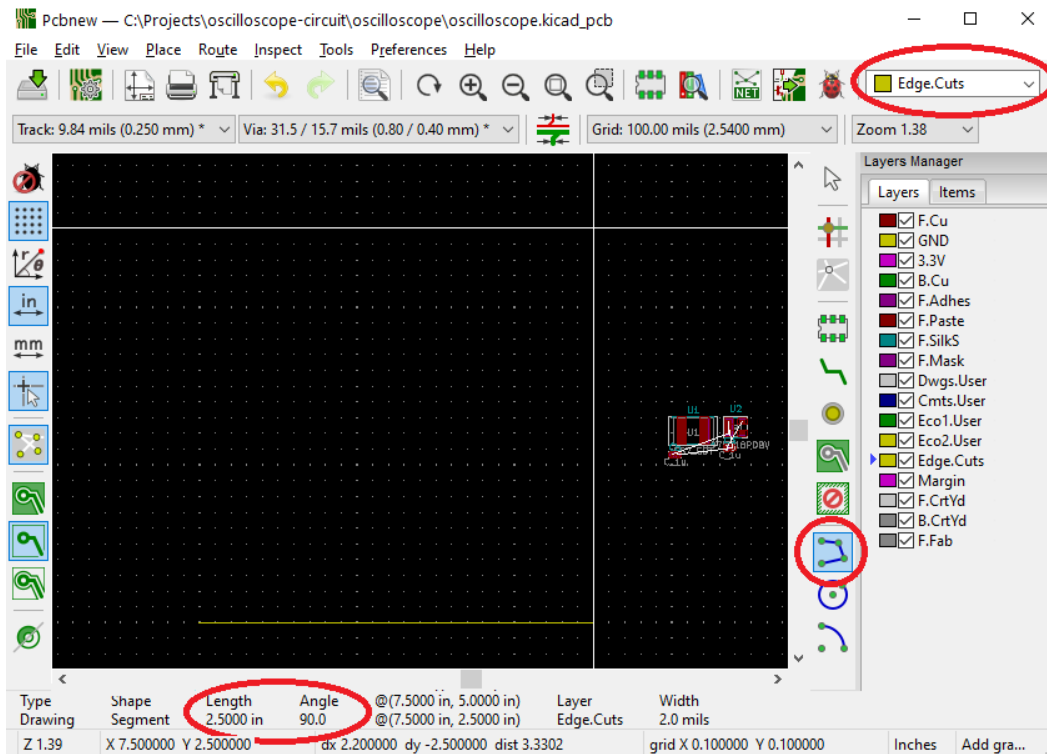


Figure 6.22 – Drawing the circuit board outline

We will now begin placing the components on the circuit board. Some rules will help organize the part placement process:

- If you did a good job organizing your schematic, the location of the parts in the diagram should give a good starting point for placing components on the board. In general, components that have direct connections to other components should be placed close together.
- Place the larger integrated circuits first. Leave some space around each of them for components, such as capacitors, that are connected to the device.
- Components that process analog signals should be placed as far as possible from digital components.
- High-speed differential signal lines should be kept as short as possible. Devices that use these signals should be located close to the connectors or circuit components at the other end of the communication path.

Although we will not be laying out the complete circuit for the oscilloscope project in this section, it is helpful to know that the connectors to the Arty board will be placed at the bottom edge of the PCB layout as it appears onscreen, and the BNC connector that receives the analog input signal will be located at the center of the top of the PCB layout.

Perform the following steps to lay out the +1.8 V power supply circuit and the GDT:

1. Change the grid to 25 mils.
2. Move the U1 and U2 integrated circuits to appropriate locations within the PCB edge boundaries. A suitable location allows sufficient space around the component for placing other items, such as resistors and capacitors, that it requires, while keeping it close to other components that provide its input signals and receive output signals from it.
3. Move the capacitors associated with each component to locations near the device. After clicking each component to select it, you can reposition it with the cursor and rotate it by pressing the *R* key. Be sure to select the entire component and not just a single pad before you move it. You may have to click a few times at different locations to highlight the entire component.

After completing these steps, the PCB layout should appear similar to the following figure:

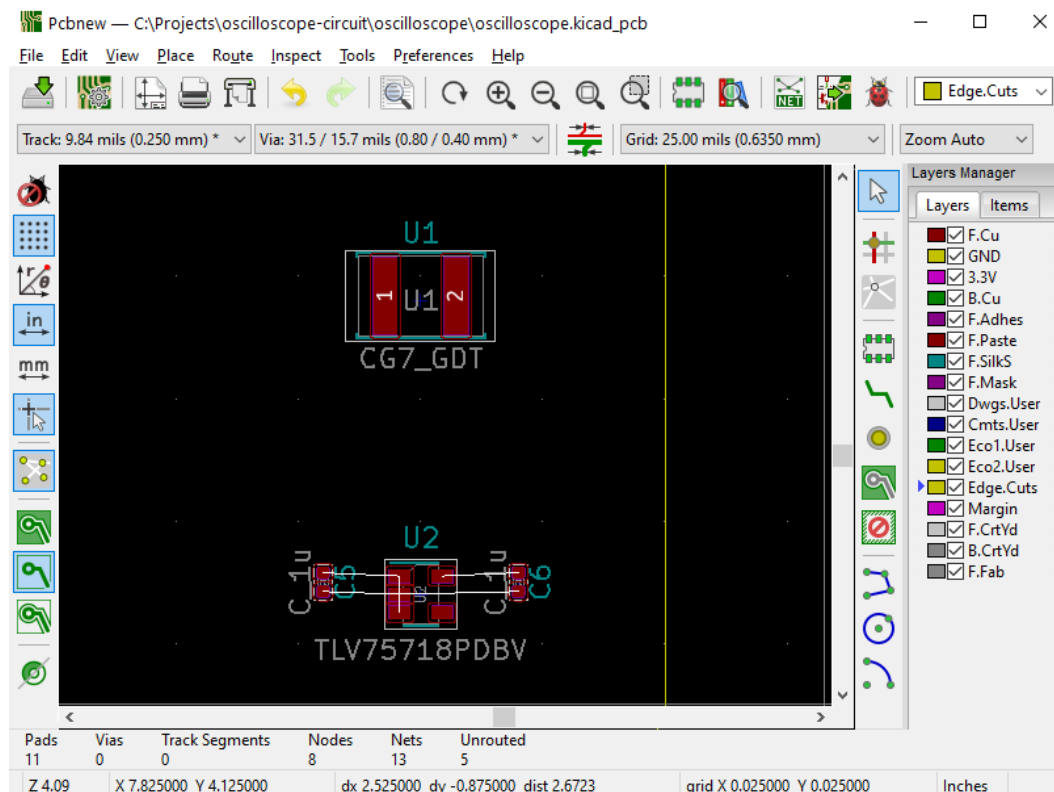


Figure 6.23 – Initial component placement

We will next define the board layer set. While the circuit we are working with so far in this example is very simple and would work well as a two-layer PCB, for the oscilloscope project we will instead use a four-layer board. The two inner layers will be power (+3.3 V) and **GND** planes. This arrangement provides low-impedance power and ground connections at all locations on the PCB and, perhaps more importantly, provides a **GND** plane beneath the high-speed differential signal traces, which is necessary to maintain signal integrity.

We will define the four layers as a top layer upon which the components will be mounted, a **GND** plane below that, a **3.3V** power plane below that, and finally a bottom layer for routing traces using vias. Each via is an electrical connection between layers created by drilling a hole and providing a conductive metal connection between the layers.

From the PCB editor **File** menu, select **Board Setup...** In the drop-down list at the top of the **Board Setup** dialog, select **Four layers, parts on front**. Change the names of the two inner layers to **GND** and **3.3V** and set the two inner layer types to **power plane** as shown in the following figure:

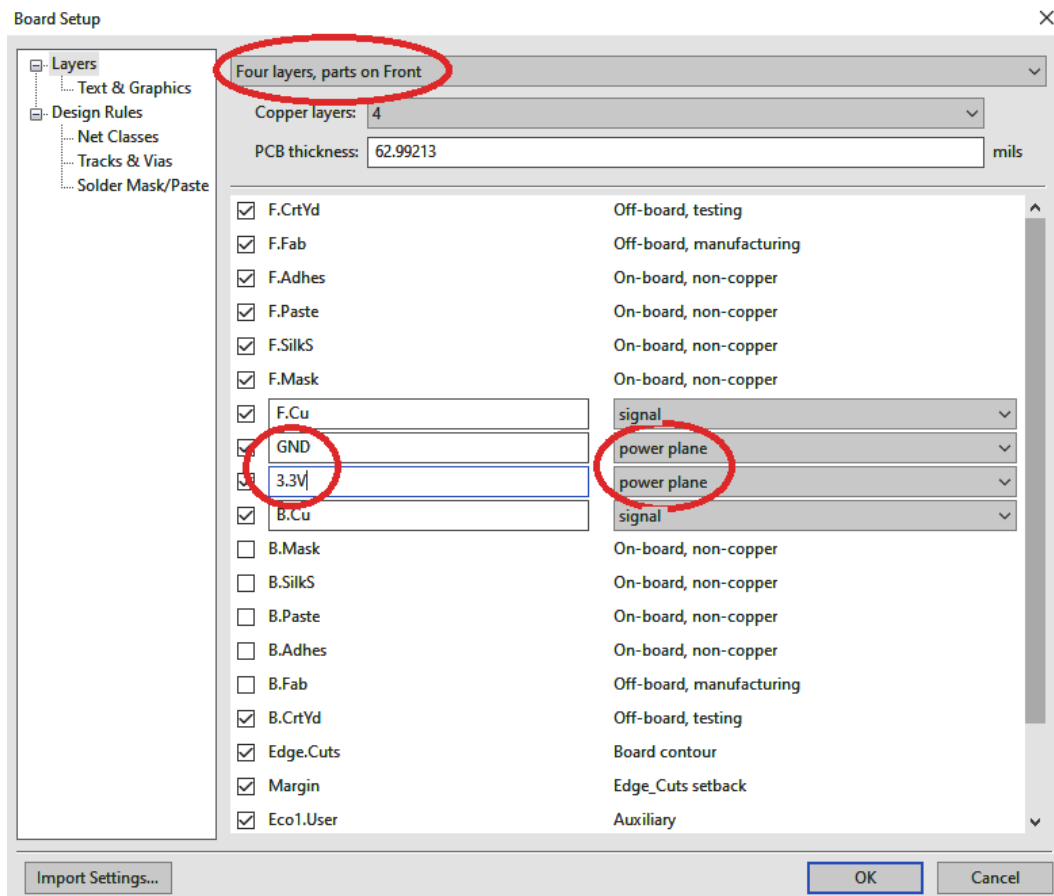


Figure 6.24 – Board Setup dialog

We will be using a **GND** fill on the top, second, and bottom layers, and a **+3.3V** fill on the third layer. A fill layer places copper at all locations in the layer except where traces, vias, or other circuit elements prevent placement. By using a **GND** fill on the top layer, all of the **GND** connections to our components will be made automatically to the fill, which avoids the need to run separate traces for those connections.

Perform the following steps to create the filled zones:

1. Set the **Grid** dropdown to 100 mils.
2. Select the **F.Cu** (front copper) layer in the dropdown in the toolbar, or just press the *Page Up* key on your keyboard to select the top layer.
3. Select the *Add filled zones* icon in the right-side toolbar. This icon looks like a PCB trace with a green background.
4. Click on the bottom-left corner of the PCB outline. This will bring up a dialog requesting a selection of the net to use for the filled zone. Select **GND** and click **OK**.
5. Click around each of the corners of the board and click again on the bottom-left corner to complete the rectangle.
6. Repeat the process for each of the remaining layers, selecting **GND** as the net for layers 2 and 4 and **+3.3 V** for layer 3.
7. Return to the top layer by pressing the *Page Up* key and reset the grid size to 25 mils.

We are now ready to draw circuit traces. Select the *Route tracks* icon in the right-side toolbar. This icon is a three-segment green line.

Each trace in need of routing is represented by a white line between the two endpoints of the connection. While in track routing mode, click on a pad at one end of each connection and draw the trace between them. If you need to create corners, click at the corner location and continue drawing. KiCad will only allow you to draw traces where there is sufficient space available and other design rules are satisfied. Complete the trace by clicking the pad at the endpoint. Press *Esc* to leave track routing mode.

For each connection to the **+3.3 V** plane, place a via near the pad location. The via icon is in the right-side toolbar and appears as a filled yellow circle with a green circle inside. After placing each via, double-click it and select the **+3V3** net.

After making all the connections, press the *B* key to fill all zones. This should complete the circuit connections to the **GND** plane on the top layer.

Pin 2 on the TLV757P will continue to show a white line, indicating it is unconnected. Click the *Show filled areas in zones* icon on the left toolbar (which looks the same as the *Add filled zones* icon in the right-side toolbar) to display the filled zones. Although there is a connection between pin 2 and the ground plane, it is too narrow for the design rules to accept. This is shown in the following figure:

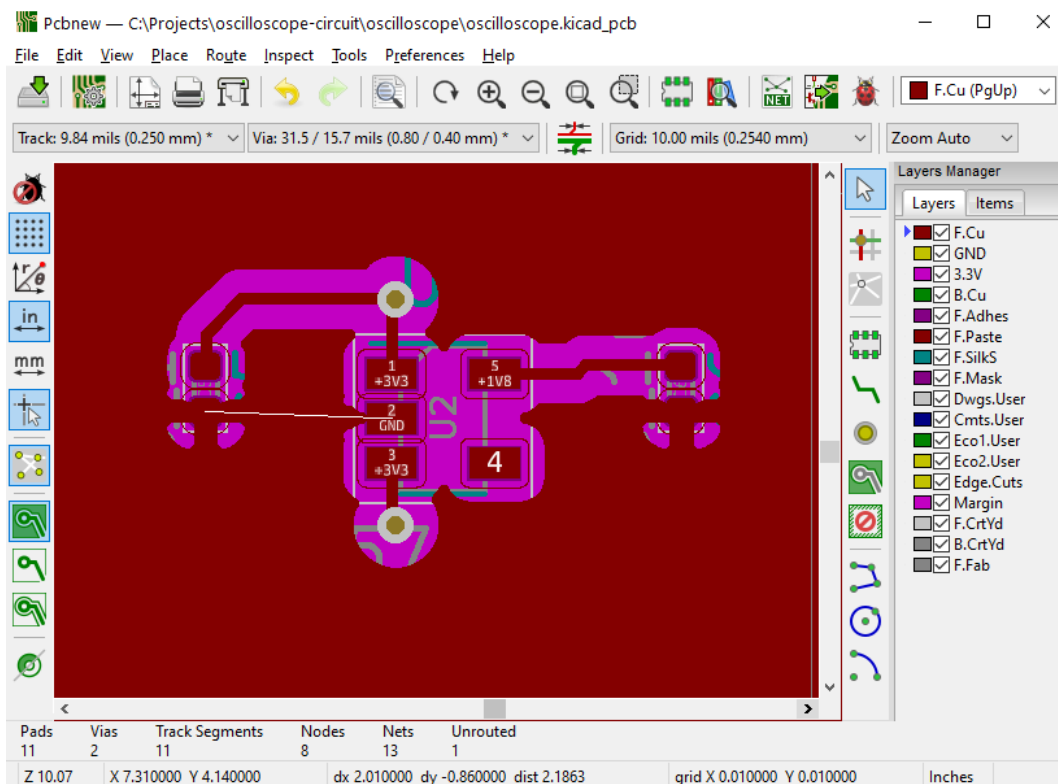


Figure 6.25 – Insufficient GND connection

To correct this, simply draw a trace from pin 2 a short distance into the **GND** plane. After adding this trace, the status bar should indicate zero unrouted traces.

Click the *Perform design rules check* icon in the toolbar, which has a ladybug on it. Click the **Run DRC** button in the ensuing dialog to perform the checks. The results should indicate zero problems and zero unconnected items. If there are any problems, the list will indicate the problem areas.

KiCad includes a 3D viewer that displays a representation of the manufactured PCB populated with components based on the PCB layout.

To display a 3D image of the board, on the **View** menu, select **3D Viewer**. This brings up a separate window with a 3D display of the PCB. Using the controls in this window, you can rotate and view the board from different angles. The 3D view of our circuit is shown in the following figure:

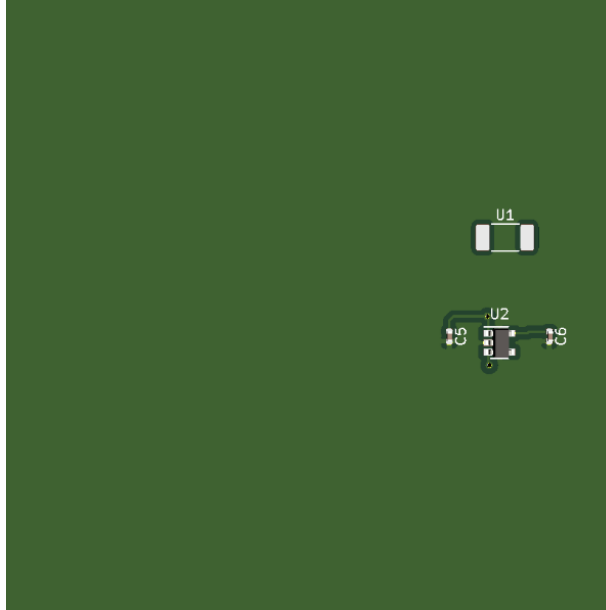


Figure 6.26 – 3D view of the PCB with components

This completes our introduction to KiCad. This example has covered the essential KiCad features you must know to understand the schematic diagram and PCB layout for the oscilloscope project.

The most significant PCB layout feature we did not cover in this example that is needed for the oscilloscope project is the use of differential pair traces. For trace pairs with appropriate names (with one signal name ending in + and the other ending in -), you can select **Route | Differential Pair**. While routing a differential pair, KiCad will continuously apply design rules and only allow you to route the differential pair along a track that satisfies the rules for trace pair routing. After routing a differential pair, you can increase the length of the shorter trace in the pair to match the longer trace by selecting **Tune Differential Pair Skew/Phase** in the **Route** menu. This ensures the two signals take the same time to travel through the traces and arrive simultaneously at the destination.

With a complete PCB layout that passes the design rule checks, we are ready to hand the circuit design off to a PCB manufacturer to produce prototype circuit boards.

Prototyping the circuit board

A number of low-cost PCB prototype board vendors offer services to hobbyists and other small-scale clients. Some vendors require a set of Gerber files in industry-standard formats for production. Others will directly accept a KiCad project as their input, which saves some minor steps.

In this example, we will use OSH Park as the vendor. You can place orders with OSH Park at <https://oshpark.com/>. OSH Park produces prototype PCBs in a distinctive purple color directly from KiCad project files. The following figure shows the board top for the fifth revision of the oscilloscope project PCB:

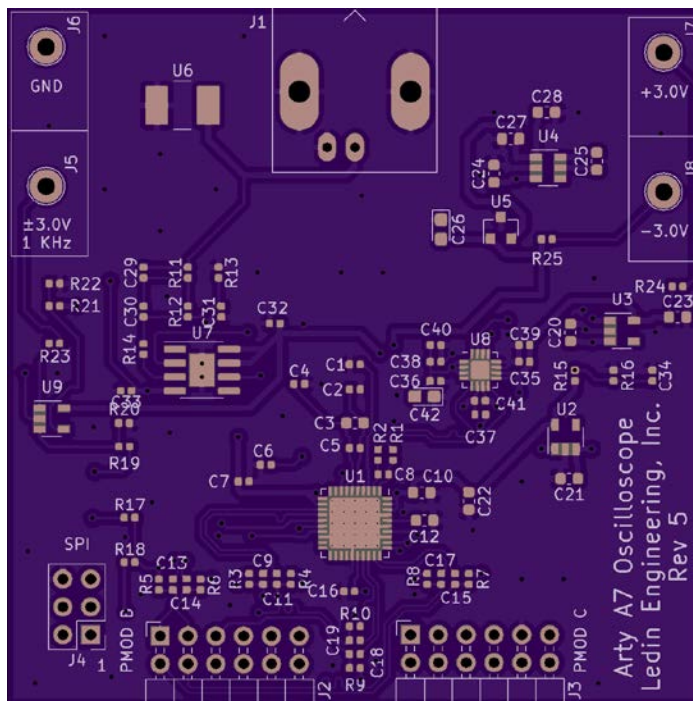


Figure 6.27 – OSH Park PCB rendering for the oscilloscope project

At the time of this writing, OSH Park charges \$10.00 per square inch to produce three copies of a four-layer PCB. Since our board is 2.5" x 2.5", the cost for three boards is \$62.50 plus sales tax, and comes with a free shipping option.

When ordering a new PCB, it is a good idea to get a matching solder paste stencil. The solder paste stencil will be used to apply solder paste to all of the metal pads on the PCB in a single operation. Solder stencils are typically constructed from polyimide film or stainless steel. Steel is more durable for large production runs, though it is also more costly. For our purposes, polyimide film is appropriate in a thickness of 3 mils (0.003"). When ordering PCBs from OSH Park, you can transfer your PCB information to OSH Stencils at <https://www.oshstencils.com> and order a stencil there. A polyimide stencil for the oscilloscope PCB costs \$15.62 including sales tax and shipping cost.

Summary

This chapter introduced the open source KiCad electronics design and automation suite and provided some examples of its use. You learned how to turn your circuit board design into a PCB prototype at a very reasonable cost. The chapter included examples of schematic entry and PCB layout for the oscilloscope project you will assemble in the next chapter.

After completing this chapter, you have downloaded and installed KiCad, learned basic procedures in KiCad, learned how to create circuit schematics in KiCad, learned how to develop circuit board layouts in KiCad, and worked through a portion of the circuit board design example for the digital oscilloscope project.

The next chapter will introduce the equipment and techniques involved in assembling high-performance digital devices using surface-mount and through-hole electronic components.

7

Building High-Performance Digital Circuits

This chapter presents the processes and techniques involved in assembling prototype high-performance digital circuits using surface-mount and through-hole electronic components. A recommended set of tools will be identified, including a soldering station, a magnifier or microscope, and tweezers for handling tiny parts. The reflow soldering process will also be introduced, along with descriptions of some low-cost options for implementing a small-scale reflow capability.

Preparations for circuit assembly and the techniques of soldering by hand and with the reflow process will be presented in a manner that allows them to be applied to a wide variety of projects. Once soldering is complete, the board must be cleaned and thoroughly inspected to ensure all intended connections are intact and that no unintended solder connections exist before applying power.

After completing this chapter, you will have learned how to assemble digital circuit boards. You will understand the tools and techniques required for circuit board assembly, as well as the steps involved in preparing for soldering. You will have learned how to solder surface-mount and through-hole components to the circuit board by hand and using a reflow system, and will also know how to clean the assembled board and thoroughly inspect it.

We will cover the following topics in this chapter:

- Circuit board assembly tools and procedures
- Preparing for assembly and placing parts
- Reflow soldering and hand soldering
- Post-assembly board cleaning and inspection

Let's get started!

Technical requirements

The files for this chapter are available at <https://github.com/PacktPublishing/Architecting-High-Performance-Embedded-Systems>.

Circuit board assembly tools and procedures

As a developer of a prototype digital device, you can construct a professional-looking, high-performance circuit board using surface-mount technology with a suitable set of tools and a bit of practice. This section will cover the needs of individual developers who are assembling a small number of circuit boards, typically one at a time. For manufacturers who need larger numbers of boards, ranging from dozens to hundreds or more, it is likely to be more appropriate, though more costly, to delegate this work to a company that specializes in PCB assembly.

Optical magnification

We will be assembling some very small components on the PCB. Many of the resistors and capacitors on our project circuit board use the 0402 package, with dimensions of 1.0 mm x 0.5 mm. To allow you to appreciate the size of these components, the following photo is of a resistor in the 0402 package size beside a grain of jasmine rice:



Figure 7.1 – Surface mount resistor beside a grain of rice

It may feel intimidating to work with such tiny components, but with a modicum of patience and practice, most people should be able to construct functioning circuit boards containing these components. Even if your eyesight is not the best, and if your hands shake a bit when you try to hold them perfectly still, such limitations can often be overcome.

The use of a magnifier or microscope allows you to get a view that is as close up as you desire, though you should not overdo it by using too much magnification. While many developers with reasonably good eyesight are happy to assemble PCBs with an inexpensive magnifying lens on a hands-free mount, or with no magnification at all, for small-component circuit assembly, I prefer to use a stereo microscope set at a low magnification. An example of this type of microscope can be seen in the following figure:



Figure 7.2 – Stereo microscope suitable for SMT soldering

When using a microscope, you should adjust the magnification to a level that allows a clear view of components the size of the resistor shown in *Figure 7.1*, while at the same time providing a sufficiently wide field of view of the surrounding board real estate. This allows you to remain oriented to the location on the PCB as you bring components in for placement using tweezers.

Any shaking that may occur when you're using tweezers can be reduced by providing a stable place to rest your hand while you're placing components. This hand rest can be constructed from books, bean bags, or other items that provide a comfortable and solid surface for your hand. Some circuit builders find it helpful to hold their tweezer hand with the other hand while placing smaller components.

It is not critical to place components at the exact location with perfect alignment on the first attempt. Placing the part in roughly the correct location is good enough initially. The part can then be moved and rotated by nudging it with tweezers or other appropriate small tools. Even when hand soldering, it is not necessary for the components to be in perfect alignment. It is good enough if you can solidly solder each of the connection points on the component to the board while avoiding problems such as **solder bridges**, which are unintended connections between signals on a PCB. Solder bridges are generally caused by applying too much solder when making connections. We'll learn how to fix solder bridges later in this chapter.

Tweezers

You need at least one good pair of precision tweezers for placing components on the circuit board and, when hand soldering, holding components in place while soldering the first connection. You should be able to find a precision tweezers set for a few dollars that includes a variety of tip orientations. The following photo is of a suitable pair of tweezers for electronic assembly:



Figure 7.3 – Tweezers suitable for electronic assembly

A pair of tweezers with a tip curved about 30°–45° from the body works well when placing components under a microscope. The angle of the tip allows you to place components straight downward while keeping your hand off to the side, away from the microscope hardware and out of your line of sight.

Flux

Oxidation is the enemy of good solder joints. **Oxidation** is a chemical process similar to the rusting of iron in that it produces a layer of poorly conductive material on a soldering iron or circuit component. When soldering, it is important to take a few steps to prevent oxidation from forming and degrading the quality of your solder joints. Fortunately, this is easy to do.

To prevent issues related to oxidation during soldering, it is important to keep the soldering iron tip clean, as described later in this section, and to use flux properly during soldering. **Flux** is a chemical compound that is nonconductive and inert at room temperature. When heated during soldering, flux performs three functions:

- It cleans oxidized deposits from the metal surfaces being soldered.
- It forms a protective barrier, preventing air from reaching the melted solder and the surfaces being soldered and causing oxidation.
- It performs a wetting function, allowing the liquid solder to flow freely over the metal surfaces. **Wetting** refers to the flow of liquid solder to all parts of a joint.

If you were to try soldering SMT components with no flux at all, you would find it very difficult to get a solid connection between the component and the board. This is due both to the difficulty of getting the solder to bond to the oxidized metal surfaces and to the poor flow characteristics of melted solder in the absence of flux.

Some types of solder wire have a hollow core filled with flux. When using this type of solder, it is important to ensure you use the soldering iron to heat the component and the PCB, then apply the solder to the heated metal until it melts and forms a joint. If, instead, you melt a glob of solder onto the tip of the iron and then attempt to form a joint, you will observe smoke drifting away as soon as the solder is on the tip. This is the flux burning off. If you use this approach, you convert flux-core solder into fluxless solder that is incapable of forming a good joint.

As an alternative to flux-core solder, it is possible to apply flux to the PCB with a felt-tip flux pen or directly as a liquid from a bottle. When soldering a PCB that has a pre-applied flux coating, you will find it is easy to produce good-quality joints. It does not hurt to have too much flux on the board, and too much flux is far better than having too little.

Some types of flux remain corrosive after soldering is complete and must be removed by a cleaning process. Another category of flux is called *no clean*, meaning the flux residue can be left in place once assembly is complete. Flux residue can be visually unappealing and sticky. Fortunately, it is easy to remove leftover flux, as we will see later in this chapter.

Solder

Solder for electronic assembly comes in two general types: tin/lead and lead-free. The tin/lead variant, which is commonly used for electronic assembly, is 63% tin and 37% lead by weight and has a melting point of 361°F (183°C). The nice features of tin/lead solder include its relatively low melting temperature and the ease with which you can produce high-quality solder joints. Its not-so-nice attribute is that lead is poisonous, even in small quantities.

It is possible to work safely with lead solder, as long as you set up an appropriate work environment and use reasonable care when soldering and cleaning up afterward. When working with solder containing lead, you should avoid inhaling fumes during soldering and wash your hands thoroughly after handling the solder.

You can avoid inhaling fumes during soldering with the assistance of a fume extractor or a fan. A fume extractor has an inlet that must be placed near the soldering area to suck in air, pulling the smoke produced during soldering into the device. The smoky air passes through a filter that absorbs the smoke residue before passing the clean air back into the room or outside via an exhaust system. A fan, on the other hand, simply blows the smoke around the room. Clearly, a fume extractor is better because it removes the smoke residue, rather than just spreading it around, though an extractor will generally cost more than a fan. A fan is better than nothing, because it will at least prevent the smoke from rising directly into your face.

Lead-free solder is available in a few different formulations. One readily available formula used in electronic assembly is 99.3% tin and 0.7% copper. This tin/copper alloy has a melting point of 441°F (227°C), significantly higher than the melting point of tin/lead solder, making lead-free solder somewhat more difficult to work with. The higher melting point of lead-free solder may also produce a greater quantity of flux fumes due to the higher soldering temperature. Even though lead-free solder avoids the hazard of lead poisoning, it is still unhealthy to inhale the fumes it gives off. Proper ventilation is essential for soldering in all cases. If you are manufacturing an electronic product for commercial sale, the use of lead-free solder may be mandatory.

When selecting solder wire, it is important to select an appropriate thickness. Because we are working with very small SMT components, it is necessary to use thin solder wire. If the wire is too thick, it becomes very difficult to melt an appropriate quantity of solder at the desired location. For the sizes of the components we will be soldering, the solder wire thickness should be no greater than 0.032" (0.81 mm). I prefer to use 0.020" (0.51 mm) rosin core solder for hand soldering SMT components. **Rosin** is a particular type of flux made from tree sap that is commonly available in hollow core solder wire. The following photo is of a spool of 0.020" rosin core solder:



Figure 7.4 – Spool of 0.020" rosin core solder wire

When using solder wire, it is best to pull about 1 foot of wire from the spool and use the heated tip of the soldering iron to cut the wire segment from the spool. The piece of wire can then be used to perform delicate soldering until it has been reduced to a length too short to hold easily for further soldering. Be sure to handle unused solder wire and soldered items as hazardous waste and dispose of them appropriately.

Electrostatic discharge protection

Many of the integrated circuits and other electronic components we will be assembling are sensitive to **electrostatic discharge (ESD)**. ESD occurs when two electrically charged bodies either come into contact or move close enough together that a spark can jump the gap between them. When this happens, a large electrical current can flow for a very brief period of time, damaging or destroying sensitive electronic components.

Because ESD is so potentially harmful to the success of your circuit construction project, it is worth taking precautions to prevent it from causing problems. You can take a few steps to minimize the risk of ESD damage to components while assembling circuits.

The general goal of ESD protection during circuit assembly is to prevent the buildup of electrical charge on you and on the items you're working with during circuit assembly. Follow these steps to reduce the risk of harmful ESD:

1. The first step in ESD protection is to use an ESD-safe mat as your work surface. An ESD mat has a high electrical resistance, but it is not an insulator. By connecting the ESD mat to electrical ground (which is available at the center screw on standard electrical power outlets), you can ensure that any electrical charge on the components and tools you are working with will dissipate when those items are placed on the mat. ESD mats are widely available at reasonable prices.
2. The second step in ESD protection is to prevent electrical charge from building up on your body. An ESD wristband with a wire connected to ground will dissipate charge from your body and prevent you from zapping components when you touch them.

If your work area has synthetic carpeting or another floor covering associated with electrostatic charge buildup, you can put down an ESD-safe floor mat to reduce charge buildup as you move about the area.

When you receive electronic parts you have ordered, you will see they arrive in ESD-safe protective packaging. It is critical to only open the packaging and remove the components in an ESD-safe environment while you are wearing a grounded wrist strap. It is generally best to leave all the components inside their packaging until you are ready to assemble them on to the board.

Hand soldering

We will cover two basic approaches for constructing prototype circuit boards: hand soldering and reflow soldering. This section will discuss hand soldering; reflow soldering will be covered in a later section. As you will see, reflow soldering is the preferred approach when working with large numbers of SMT components. Although we will recommend the use of reflow soldering where possible, in most circuit projects, there is also a need for some hand soldering to add components such as connectors for power and communication paths to external components. Hand soldering is also the way we repair any problems that arise with component placement and connections during reflow soldering. If you lack the tools necessary for reflow soldering, hand soldering is available as an alternative.

Hand soldering involves the use of a handheld soldering iron and other handheld tools, including a hot air gun, solder, and tweezers. For the work we will be doing, it is best to use a soldering station that combines a handheld soldering iron with a handheld hot air gun. This type of soldering station is often called a **rework station**, which refers to a station that can be used to disassemble and repair circuit boards, in addition to performing circuit assembly. The following is an example of a rework station:



Figure 7.5 – Solder station with an iron and hot air gun

The rework station shown in the preceding figure provides digital displays of the temperature of the soldering iron and the hot air gun. An additional control is provided to adjust the airflow through the hot air gun. Most soldering stations of this type include a variety of tip shape options for the soldering iron and a selection of nozzle shapes and sizes for the hot air gun. Because we are working with very small components, the best tip for the soldering iron is most likely to be the tip that comes to the smallest point. This enables heat transfer to the precise location where it is desired without heating, and possibly damaging, other parts of the circuit.

Accurate temperature control is critical when soldering surface mount components. The least expensive soldering irons do not provide a mechanism for monitoring or adjusting temperature, so they should be avoided when performing PCB assembly.

As a starting point, a soldering iron temperature that's 30–50°F (20–30°C) above the melting point of the type of solder you are using is suitable for soldering small components. As your skill level increases, you can increase the temperature to melt the solder faster. With higher temperatures, you will need to complete the joint and remove the iron quickly to avoid overheating and damaging the component.

When soldering, you should make frequent use of the tip cleaning pad, which is usually a sponge or a clump of coiled wire. If a sponge is available, as is the case with the solder station shown in the preceding figure, it must be kept wet; otherwise, the hot iron will burn it. The sponge should be fairly damp but not saturated with water. To clean the solder iron tip using the sponge, the iron must be fully heated. Wipe the tip across the sponge while twisting the iron to clean the full circumference of the tip. If you're using a wire coil tip cleaner, insert the hot iron tip into the coil a few times to give the wires a chance to scrape off any excess solder and oxidation.

The solder station hot air gun is useful for heating a larger area in comparison to a soldering iron. This area-heating capability can melt solder on multiple pads simultaneously, such as when removing an SMT integrated circuit from a board. Hot air can also be used to solder integrated circuits onto the board.

Some IC package types are not suitable for soldering with a soldering iron and, if a reflow soldering process is unavailable, can only be soldered using hot air. For these components, a generalized procedure is to apply a light coat of flux to the board, then use a soldering iron to apply a thin layer of solder to the pads (this is called **tinning**), before placing the IC on the pads and using hot air to solder all of the pads simultaneously. When using hot air to solder components in this manner, it is important to avoid blowing the part off-center and to avoid inadvertently desoldering and blowing away any surrounding components. For this reason, any components that must be soldered using hot air should be installed on the PCB first.

Solder wick

Sooner or later, when hand soldering, you will accidentally apply too much solder to a joint and end up with a solder bridge between pins on an IC or between closely spaced components. It is straightforward to remove excess solder using solder wick. **Solder wick** is braided copper wire intended to absorb melted solder. Just like when you're assembling components, removing excess solder works best when there is flux present on the wick and on the area containing the excess solder.

To use solder wick, first ensure there is flux present on the wick and on the area containing the excess solder. Place the clean, unused end of the wick on the excess solder, then press down on the wick with the hot soldering iron. It may take a few seconds, but, eventually, the solder will melt and flow into the wick. Once the solder has been absorbed, remove the wick and the soldering iron from the board simultaneously. If you remove the iron first, the melted solder will harden, and the wick will remain stuck to the board.

The following is an example of solder wick after being used to remove some solder:



Figure 7.6 – Solder wick containing removed solder

After using solder wick, clip off the section of wick containing the removed solder. This leaves a clean end for its next use.

Solder paste application

Reflow soldering uses **solder paste**, which is a mixture of sticky flux and microscopic balls of solder, to loosely attach a potentially large number of circuit components to their pad locations on a PCB. Once all the components have been placed onto their solder paste-covered pads, the entire PCB assembly is subjected to a heating profile that melts the solder, forming a solder joint at each pad location. The term **reflow soldering** refers to the fact that each time the solder temperature is raised above its melting point, it becomes liquid, with the corresponding ability to flow.

For the developer of a prototype circuit board, one obvious advantage of reflow soldering compared to hand soldering is that you don't need to solder every connection point between the components and the PCB. Instead, it is necessary to apply solder paste to the component pads and place the components accurately before heating the board.

One approach is to apply solder paste manually, using a syringe to dispense the material onto individual pads. It is also possible to use a stencil containing appropriately sized holes at each PCB pad location to apply solder paste to all the pads on the PCB in a single operation. Solder paste stencils are constructed using the data files produced during the PCB layout process, as discussed in the *Prototyping the circuit board* section of *Chapter 6, Designing Circuits with KiCad*.

To apply solder paste to a PCB using a stencil, it is helpful to prepare a frame that will hold the stencil in place on top of the PCB, as shown in the following figure. The frame helps ensure the holes in the stencil remain in position over the PCB pads and that when you're lifting the stencil away, the movement is straight upward rather than sideways, which may smear the solder paste. You should be able to purchase a frame when you buy the stencil, though you can make one yourself from suitable material such as unpopulated circuit boards:

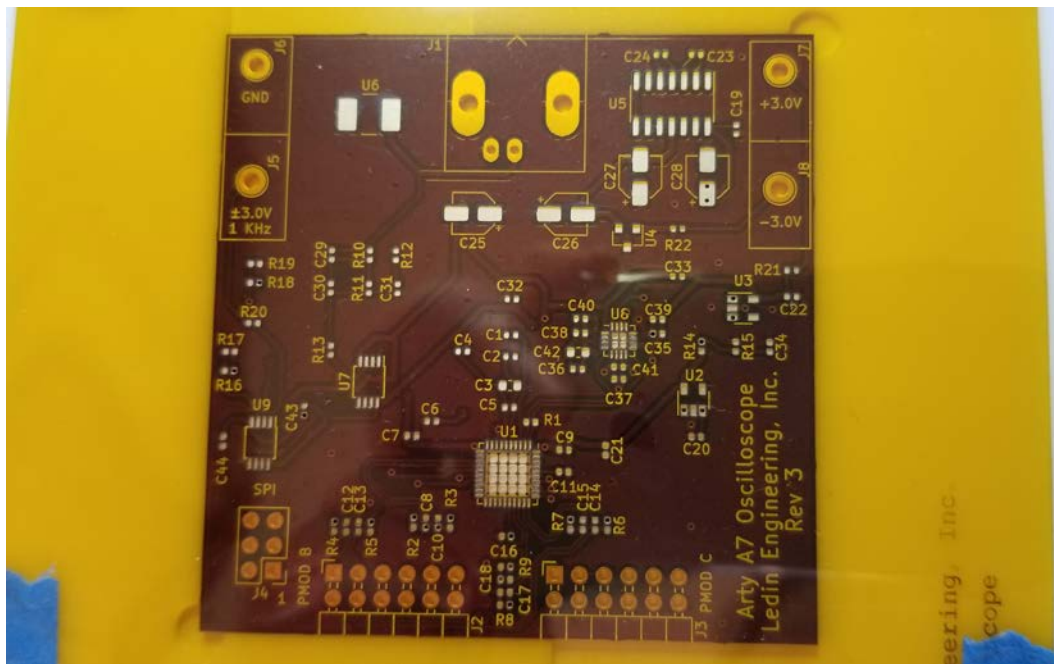


Figure 7.7 – Solder stencil attached to a frame

It is best to use solder paste purchased in a jar for working with solder stencils. By utilizing this approach, you will apply much more solder paste to your spreader than you actually need to fill in the holes in the stencil. Once stenciling is complete, you can scrape the excess paste from the stencil and put it back in the jar for later use. If, instead, you dispense the paste from a syringe, you will most likely discard any excess paste.

For spreading the paste onto the stencil, you will need a tool with a straight, flexible edge. Depending on the size of your circuit board, this tool might be anything from a credit card, to a putty knife, to a large drywall taping knife. For our project circuit board, the free credit card-sized paste spreader provided by OSH Stencils with each stencil purchase will serve our needs.

Applying the solder paste to the stencil atop the PCB in the frame can be done quickly, but it requires a bit of finesse. In particular, if you find you have missed areas of the stencil holes on the first pass, or if the application is otherwise uneven, repeated attempts to apply the paste may result in excessive buildup of solder paste under the stencil – in other words, a mess. Don't be discouraged if you need to clean off the PCB and the stencil after your first attempt and start over. You can clean the solder paste from the stencil and from the board using **isopropyl alcohol (IPA)** and a lint-free cloth.

To apply the solder paste to the stencil, perform the following steps:

1. Ensure the PCB, the frame, and the stencil are lying perfectly flat on a hard surface. The frame should be the same thickness as the PCB, and the holes in the stencil must be carefully aligned with the PCB pads. You can use removable tape such as painter's masking tape to hold the stencil in position.
2. The spreader must be wide enough to cover all the holes in the PCB while passing in a straight line across the surface.
3. Open the jar of solder paste and stir the paste thoroughly with a tool, such as a small screwdriver.
4. Use the stirring tool to load a bead of solder paste on one side of the spreader, along the straight edge.
5. Place the spreader on the stencil, with the side holding the solder paste facing toward the area to be stenciled.
6. Tilt the top of the spreader about 45° in the direction it will be moving.
7. In one smooth sweep, with continuous downward pressure, move the spreader across all the stencil holes. This should completely fill the holes with paste.

8. Turn the spreader around and prepare to make a sweep in the opposite direction. The purpose of this sweep is to scrape off the solder paste residue from the stencil's surface.
9. Tilt the spreader so that it's nearly vertical, and leaning slightly in the direction of motion. Make another smooth sweep in the opposite direction of the first sweep.
10. Scrape the solder paste residue from the spreader into the jar and seal the jar.
11. Carefully lift the stencil from the PCB.
12. Examine the PCB and verify that all the pads that should have received solder through the stencil have received it. Not all the pads should receive solder paste. Some, such as those for through-hole connectors, are not intended to be soldered during this process.

If all has gone well, you should now have a PCB with appropriate quantities of solder paste applied to all the SMT component pad locations. Feel free to congratulate yourself on this success, but be aware that by completing this step, you have started a ticking clock.

Solder paste has a stickiness, referred to as **tack**, that holds components in place once they have been positioned. Once the paste is exposed to the air, some of its chemical components begin to evaporate, reducing its stickiness. As a general rule of thumb, you should plan to wait no more than 8 hours between stenciling solder paste onto a board and completing the reflow process. In other words, you should not plan to wait overnight once you've stenciled a board before you populate it and perform the reflow.

Applying solder paste transforms the PCB into an extremely delicate item that can be easily disturbed. You must be exceptionally careful when handling the stenciled PCB and while placing components on it to ensure you do not brush against any of the paste-covered pads, as this will wipe the paste away, along with any components you may have already placed at those locations. Even the lightest contact between your hand or tweezers and the paste may have drastic consequences.

When placing components on the solder paste-covered pads, it is best to work from the center of the board outward. Whether you use a magnifier, a microscope, or no magnification at all, you must be very careful to avoid contact with the board at any point other than the target placement location for each component. As you continue populating the PCB, rotate the board so that you do not have to reach across to the far side to place any of the components.

Reflow soldering

Once all the components have been placed, the board is ready for reflow soldering. The reflow process is standardized across the industry for SMT boards and circuit components. In factories, large quantities of PCBs are soldered in succession in large reflow ovens that transport the boards through areas of varying temperature based on the intended heating profile.

For hobbyists and other small-scale prototype PCB developers, it is not practical to employ the processes and equipment used in electronic device factories for building one board at a time. The essential functionality of the reflow process equipment is to heat the PCB and the solder paste it carries to a temperature that allows the solder to melt and reflow, while avoiding problems such as overheated and damaged circuit components.

A typical reflow temperature profile can be seen in the following diagram. The profile contains four major phases: preheat, soak, reflow, and cooldown:

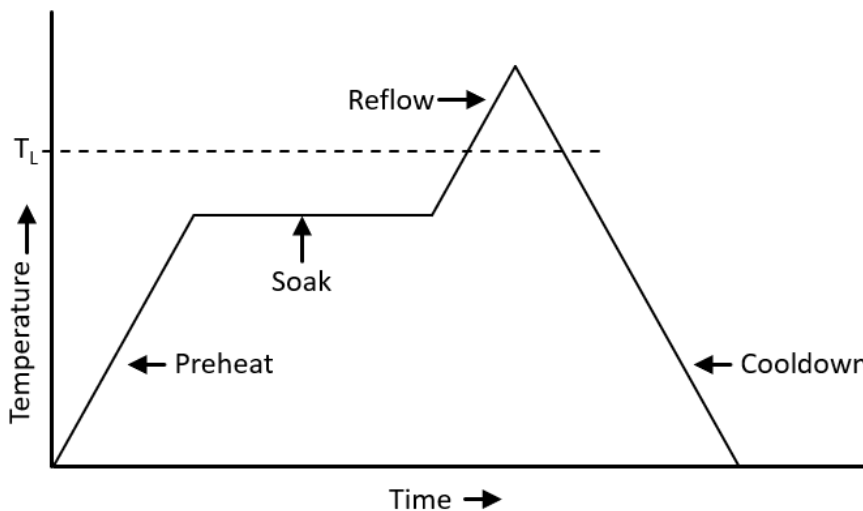


Figure 7.8 – Reflow soldering temperature profile

The purpose of each of these phases is as follows:

- **Preheat:** In this phase, the board is quickly brought to the soak temperature and the flux solvents begin evaporating.
- **Soak:** In the soak phase, the flux solvents finish evaporating, and the flux removes oxides from the surfaces to be soldered. Larger circuit components have time to approach the soak temperature, which will minimize thermal stress during the reflow phase.

- **Reflow:** The temperature increases above the **liquidus temperature** (T_L), which is the melting temperature of the solder. The temperature remains above T_L for 1–2 minutes to allow sufficient time for all the solder on the board to reflow.
- **Cooldown:** The cooldown phase returns the board to room temperature fairly quickly. A quick cooldown is desirable both for the purpose of minimizing process time and to produce stronger solder joints compared to those resulting from a slower cooling phase.

The reflow process takes about 10 to 15 minutes from beginning to end. The reflow heating profile described here represents an ideal process typically performed using sophisticated manufacturing equipment. Developers on a tight budget have developed procedures for performing reflow soldering using kitchen appliances, such as an electric hot plate and a skillet, or with a toaster oven, with varying results.

At the lowest level of sophistication, a developer can set a PCB with placed components on a skillet positioned on a hot plate and turn on the heat. By watching the solder paste closely as the board heats, you can determine when the solder has melted and remove the board from the heat. While some developers claim this technique produces consistently successful results, it is not a well-controlled process.

Other developers have used toaster ovens to perform reflow soldering with temperature indicator material. Temperature indicators allow you to place markings on a PCB that change color when the target temperature is reached. The toaster oven method relies on watching for the indicator to change color through the window of the oven door and removing the board from the heat when the appropriate temperature has been reached.

Various modifications for standard hot plates and toaster ovens to improve their reflow processing capabilities have been developed by talented individuals who retrofit these devices with temperature monitoring and control systems. These modified kitchen appliances are capable of implementing temperature profiles similar to the one shown in the preceding diagram. An internet search for phrases such as `toaster oven reflow` or `hot plate reflow` will provide you with more information about these efforts.

You can build your own modified hot plate or toaster oven, or you can even purchase a small, dedicated reflow oven for under \$300 from a variety of sources. If you use a toaster oven for reflow soldering, you must never use it to cook food. This is because the oven will become contaminated with flux residue and, if you use lead solder, lead residue as well.

Regardless of the type of reflow system you end up using, the goal is to create good connections at every solder joint on the board. The following photo shows some solder joints that have been produced in a low-cost dedicated reflow oven with hand-placed 0402 and 0603 capacitors using stencil-applied solder paste:



Figure 7.9 – Reflow soldered 0402 package capacitors

If you do not have a reflow oven available, the circuit board for the oscilloscope project can be hand-soldered, with the exception of the U1 and U8 integrated circuits. Both of these components require the back of the device to be connected to board ground, in addition to the pins around the chip perimeter. It may be possible to solder these devices using hot air, as described earlier in this chapter.

Before we move on, please review the following soldering safety recommendations.

Soldering safety tips

Be sure to keep these points in mind while soldering:

- Wear safety glasses when soldering. It is possible for hot solder to splatter if it comes into contact with moisture.
- Avoid letting the soldering iron, the hot air gun, or the air exiting the air gun come into contact with people, electrical cords, or anything else you don't want melted or burned.
- Ensure the soldering iron cleaning sponge is sufficiently wet before turning the iron on.
- Turn the soldering iron and hot air gun off immediately after you've finished using them.
- Remember that the iron and gun remain hot for some time after being turned off, and that the circuit boards and components are very hot immediately after being soldered.

- When you put down the soldering iron and hot air gun, be sure to place the tool in the appropriate holder. Do not lay a hot tool on your workspace surface.
- Use a fume extractor, or at least use a fan, to blow the fumes away from your face.
- Do not wear loose-fitting clothing or jewelry that might come into contact with the soldering iron or your circuit components.
- Wash your hands thoroughly with soap and water after handling lead solder.
- Do not reuse a toaster oven that has been used for reflow soldering to cook food.
- Ensure small children and pets cannot gain access to a work area containing hot tools and lead solder.
- Have a first aid kit available. You are likely to burn yourself at least once before you master the ability to handle a soldering iron safely.

The recommended approach for assembling the oscilloscope project circuit board is reflow soldering, using the heating system of your choice. The steps for preparing and assembling this board are provided in the next section.

Preparing for assembly and placing parts

When the time comes to assemble a circuit board, you should begin by confirming you have everything you need to finish the job. When you reach this point, you will have a bare circuit board plus all of the circuit components you will need, as well as the tools and consumables needed for applying solder paste and performing reflow soldering.

It is helpful to purchase extra circuit components in case you lose or damage any of them during assembly. This is an easy thing to do for inexpensive parts such as SMT resistors and capacitors. For more expensive components, typically integrated circuits, you will have to decide how many spares your budget can tolerate.

The task of placing a large number of components on a board can be quite tedious and error-prone. Your work area should be clear of any obstructions and distractions that might interfere with your ability to successfully populate the board.

It is critically important to ensure the correct component is placed at each location on the PCB. This may seem obvious, but because many SMT resistors and capacitors have no markings indicating their resistance or capacitance, if devices with different values get mixed together, the only way you can determine which is which is by measuring their resistance or capacitance. This is possible, but it is far better to avoid this type of problem in the first place. This work is hard enough already.

The PCB silkscreen layer shows the annotated number of each resistor or capacitor, but it does not identify the resistance or capacitance of the part. You can examine the circuit schematic diagram to identify the resistance or capacitance of each part, but this is not the easiest way to find component information as you are placing parts on the board.

KiCad can create a text file called the **bill of materials (BOM)** that lists the circuit components, along with their annotated reference numbers and label information indicating the resistance and capacitance values. This function is available in the **Pcbnew** application's **File** menu. Select **Fabrication Outputs** and then **BOM File...** The resulting output file is in **comma-separated values (CSV)** format with semicolon separators. You can import this file into a spreadsheet application and use it as a guide as you place components. I find that the best way to do this is to print out a page with the BOM and cross off each component once it has been placed. This allows me to deviate from the sequential part number order and, for example, place all the 0.1 μF capacitors before moving on to a different component type.

Even with the BOM available on paper, it can still be challenging to quickly find where each component goes on the board. It is helpful to open the PCB layout in the **Pcbnew** application while placing the components. You can use the **Find...** function on the **Edit** menu, or just press *Ctrl + F*, and search for component identifiers. For example, typing *R20* in the **Search for** dialog that *Ctrl + F* brings up will highlight and place the cursor on the R20 resistor on the board layout, showing you where to find the part on the board you are assembling.

The following checklist will help you prepare for board assembly:

1. The PCB must be ready for assembly. Any undesired tabs or protrusions from the PCB manufacturing process must be removed and the board must be completely clean and dry.
2. All circuit components are available and organized so that you can find them easily. This includes a sufficient quantity of resistors and capacitors of all values needed. ESD-sensitive devices should not be removed from their protective packaging until you are ready to place them. In this phase, we are only installing surface-mount components. Other board components, such as through-hole connectors, will be installed once reflow soldering is complete.
3. A staging area is available to hold parts once you've removed them from their packaging. This area should be light-colored to make it easy for you to see the tiny components. A sheet of white printer paper works well for this purpose.

4. A microscope or magnifier, if desired, should be in a location suitable for comfortable use. Precision tweezers must be available for placing components.
5. The reflow oven or hot plate must be ready to use. You do not need to heat it up until the board is ready to bake.
6. The solder stencil must be aligned with the PCB and ready to receive solder.

Once you've completed these steps, it is time to apply solder paste to the board. Follow the steps in the *Solder paste application* section earlier in this chapter to apply the paste and inspect the results.

As we mentioned earlier, there is something of a time limit once the solder paste has been applied. This does not mean you should rush the parts placement process. Feel free to take breaks as necessary to stay relaxed and focused on the task.

As a reminder, try to maintain constant awareness of the location of your hands and tools relative to the PCB any time you are near it. It is very easy to accidentally brush against the board and wipe away solder paste and previously placed components. If this happens, stop and take a moment to evaluate the situation. If the disruption is minor, you may be able to use tools to push the paste and components back onto the proper pads. If you've lost track of which component goes where, you may need to discard resistors and capacitors and replace them with new components.

If the damage is more severe, you can decide whether you want to continue populating the board or start over. If you have more than one board, you can apply solder paste to a second board and transfer the components from the first board to the second.

Alternatively, if a portion of the board is no longer suitable for parts placement but the rest is undamaged, you may want to finish placing parts on the undamaged area and perform reflow soldering. Once reflow soldering is complete, you can hand solder the remaining components to the board. Of course, maintaining sufficient care during parts placement avoids this issue in the first place.

Once you have finished placing all the parts, congratulate yourself and take a short break. Then, perform a visual inspection and verify that all the parts have been placed and that they are in the proper locations and orientations. In particular, for any part that has a single correct orientation, such as integrated circuits and polarized capacitors, verify that the part is aligned properly.

The board is now ready for reflow soldering. The reflow soldering procedure is the subject of the next section.

At this point, the PCB sits on your assembly workspace, fully populated with SMT components. Now, it is time to turn on your reflow oven or hot plate.

Reflow soldering

Regardless of what type of reflow system you use, exercise extreme care when moving the PCB from your workspace to the oven or hot plate. If you strike the board against an object or, worse, drop it, you are likely to find that all your hard work placing the components was done for naught.

Your level of involvement during reflow soldering depends greatly on the technical capabilities of your reflow system. If you are using a stock hot plate or toaster oven, it is entirely up to you to monitor the state of the PCB as the solder heats and melts. If you remove the board from the heat too soon, you will have areas of unmelted solder, which means electrical contact will be poor to nonexistent, and parts may fall off the board. If you wait too long to remove the board from the heat, you risk overheating and damaging the components on the board.

If you have an automated oven – either a dedicated reflow oven or a toaster oven modified with temperature monitoring and control capabilities – your job is much easier. With these ovens, you simply place the board in the oven and, conceptually at least, push a button to start the process. The oven will run through a heating profile similar to the one shown in *Figure 7.8*. At the end of the cooldown phase, the board will be at a temperature not too far above room temperature.

Once the board is cool enough to handle, it is time to perform an inspection to verify that each component remained in its proper location and orientation during reflow, and that all the solder connections are smooth and shiny. You should examine locations where traces or pads are close together, such as closely spaced integrated circuit pads, for solder bridges. For IC packages that have pins reaching beneath the plastic case, it can be hard to visually determine whether the connections are good. Examine the joints from various angles to find the best view.

If you see instances of solder bridges, use solder wick to remove the excess solder. If any components are mysteriously missing or have shifted to an unsuitable orientation, or just do not appear to have a good solder connection, make note of them so that you can fix each problem during the hand soldering phase.

Hand soldering

The two objectives of the hand soldering process are to fix any problems that occurred during reflow soldering and to attach the remaining through-hole components to the PCB.

It is better to fix any problems from reflow soldering before moving on to installing the remaining board components. This is because it's easier to access the different parts of the board before the through-hole components have been installed.

Repairs after reflow

If any parts have shifted to inappropriate positions or orientations during reflow, this can be fixed with hand soldering. As always, flux should be used to improve the soldering results.

If a component has become raised or tilted, you may be able to fix it by pressing it down with tweezers or another sharp tool while you melt the solder with the iron. After doing this, ensure all the component pads have good connections. When resoldering the joints of smaller SMT resistors and capacitors, it should take just a second or two of contact with the hot iron to melt the solder.

You may need to remove some components before you can properly position them on the board. With smaller resistors and capacitors, especially if only one end is attached, you can melt the solder while lifting the part with tweezers. If a larger component, such as an integrated circuit, is in the wrong position, you will need to use hot air to remove it. Grasp the component with tweezers and attempt to lift it while simultaneously aiming the hot air directly downward on it. The intent of this operation is to release the target component from the board while you avoid melting the solder holding any of the surrounding components to the board. If that happens, the air from the gun is likely to blow the other components across the board.

When reattaching components that have been removed from the board, begin by placing a light coat of flux on the pads to be connected. Align the part and tack down one pad. Since flux is already on the pads, it is acceptable to place a small amount of solder on the tip of the iron and touch it against the part and the pad. A second or two of contact should be sufficient.

If the alignment of the reattached part looks good once the first pad has been tacked, rotate the board and tack a pad opposite from the first one. If there are additional pins, proceed with soldering each of them in the same fashion.

Installing through-hole components

Once any post-reflow repairs have been completed, it is time to solder the through-hole components to the PCB. For the oscilloscope project, this includes the BNC connector for the oscilloscope probe, the 2 x 6 pin board edge connectors that will plug into the Arty board Pmod connectors, the 2 x 3 pin connector for the SPI interface, and the four test point loops.

Compared to SMT soldering, attaching the through-hole components is less of a delicate exercise. If you have a soldering iron with a higher wattage than the iron used for SMT component soldering, you may want to use it. It may take several seconds for an iron with a fine-pointed tip to heat the component and the board to the point where the solder melts.

To attach each component to the board, first place it in position and, if necessary, use clips to hold it in place. If you use plastic clips, make sure the plastic doesn't come into contact with the metal part of the component you will be soldering first.

Start by soldering one pin of the component to the board. If you used a clip to hold it in place, remove the clip once the first pin has been soldered.

Examine the part to ensure it is properly oriented and aligned. If necessary, you can reheat the solder and realign the part. This is the advantage of soldering only one pin on the first pass.

With the part aligned properly, finish soldering the remaining pins. Take your time and ensure you use enough solder so that it fills the hole and flows through to the other side of the board. This creates a strong electrical and mechanical joint between the component and the board. This strength is critical for parts that will undergo physical stress during use, such as connectors.

The two larger posts under the BNC connector are the largest solder joints on the board. Be sure to take enough time to heat the post and use plenty of solder to fill in the pad on the back of the board.

If any of the through-hole components contain leads that protrude significantly from the back of the board, use a pair of flush-cutting diagonal cutters to trim the lead just above the solder joint.

Once all the through-hole components have been installed, take a look at the board again. Make sure all the components have been installed and oriented correctly. If everything looks good, you are ready to clean the board and perform the final inspection, prior to applying power.

Post-assembly board cleaning and inspection

It is best to clean excess flux from the board shortly after soldering is complete. If the flux has an opportunity to dry out, it can be difficult or impossible to remove the residue.

If your solder paste is of the *no-clean* variety, you may choose not to remove the residue. No-clean solder paste leaves behind a limited quantity of flux residue, and this material is non-corrosive. It may be more difficult to remove the residue of this type of flux than to remove rosin flux.

Rosin flux should be cleaned from the board after soldering. This includes work done with rosin core solder wire and with liquid or pen-applied flux. The reasons why cleaning is necessary are as follows:

- *Flux residue is visually unappealing.* Rosin flux residue is a clear, shiny material with a yellow color that users of the device will notice. Customers purchasing the product may raise concerns about the quality of construction if it is present.
- *The residue is sticky.* This is not just a problem for fingers touching the board. If a metallic object comes into contact with the sticky material, it may attach itself and produce a short circuit that damages the board.
- *Flux residue makes inspection more difficult.* If the residue covers solder joints, it can be hard to examine the quality of the work.
- *Rosin flux residue is acidic.* If not removed, the residue can lead to corrosion and eventual failure of the board.

In general, each time soldering is performed on a board using rosin flux, it is necessary to clean the board afterward. When you're reflow soldering with no-clean flux, cleaning the board is optional.

Flux removal

Cleaning flux from the board is a straightforward procedure. Although several methods can be used, including the use of a dedicated washing machine, a simple method is to use a toothbrush and 91% IPA.

A variety of flux cleaning chemicals are also available that may do a better job than IPA. For board prototyping purposes, removing enough flux to make the board look and feel clean is our objective. For commercial applications, particularly in situations where safety and long-term reliability are critical, more stringent cleaning procedures are appropriate.

When working with IPA and flux residue, you should wear rubber gloves and safety glasses. Be sure to work in a well-ventilated area. IPA is highly flammable, so avoid working around heat, flames, or anything that can produce sparks.

Pour some IPA into a small, shallow dish and dip a stiff toothbrush into the liquid. Scrub the flux-coated areas of the board with a circular motion. In the first pass, ensure all the flux is covered with IPA. Let the board sit for a minute or so while the IPA dissolves the residue, then scrub again to remove the material. Repeat this process as needed until the board is clean.

When you are finished, use either compressed air to blow the remaining IPA from the board, or use a paper wipe to dry the board. Inspect the board to determine whether all the visible flux has been removed. If necessary, repeat the flux removal process to clean the remaining areas.

This cleaning method will obviously not remove flux beneath components and between closely spaced IC pins. For our purposes, we will not concern ourselves with the small quantity of flux remaining in these areas.

Post-assembly visual inspection

Once the board has been assembled and cleaned, it is time to perform a final detailed inspection. With the flux residue removed, it will be easier to closely examine each solder joint and identify whether any repairs are needed. The most important goal of this inspection is to identify any problems that could result in a short circuit, potentially damaging the components on the board. It may also be possible to cause damage to external components, which in our case includes the Arty board, so we want to minimize the possibility of such problems.

To perform the inspection, use a magnifier or microscope and go over each solder joint methodically, examining it for proper assembly and verifying that a sufficient quantity of smooth, shiny solder is on all the joining surfaces. Some common soldering problems that occur with SMT devices are listed here:

- **Solder bridges:** As we discussed earlier, these are unintended connections between portions of the circuit. Solder bridges can form between closely spaced components, across PCB traces, and between the pins of integrated circuits. It can be difficult to visually determine whether closely spaced IC pins are improperly bridged. It may be helpful to use a multimeter to carefully test the resistance between two points and determine whether a bridge is present.

- **Insufficient wetting:** If the liquid solder failed to flow properly and cover a sufficient area of the pad and the component, it will be necessary to touch up the joint with hand soldering.
- **Cold joints:** If a solder joint appears dull and bumpy rather than smooth and shiny, this is an indication of a cold joint. The term **cold joint** refers to the fact that the solder failed to reach a temperature at which it could flow properly to form the joint. Cold joints should be reheated in the presence of flux to form a proper joint.
- **Tombstoning:** Tombstoning refers to the tendency of small SMT components, particularly resistors and capacitors, to tilt upward so that instead of touching both pads, the part stands straight up on one of the pads. This occurs when there is insufficient contact between the part and one of the pads and the surface tension of the liquid solder pulls the part upright on the other pad. Any parts exhibiting this problem must be fixed by hand soldering.

As you handle the board during inspection, continue to abide by ESD protection protocols. Work on a static-safe mat and wear a grounding wrist strap.

Following the visual inspection, and after repairing and verifying that any problems that were detected during the inspection have been resolved, it is time to perform an electrical check for shorts.

Electrical short checking

As a final check before applying power to the board, you can use a multimeter to test the power and ground connections. This check should provide you with confidence that there are no serious electrical issues remaining, such as a short between the board's power input and ground.

The power to the project board is provided through the Pmod connectors on the side of the Arty board. The following diagram shows the locations of the four +3.3 V power pins and the four ground pins, viewed while looking toward our board:

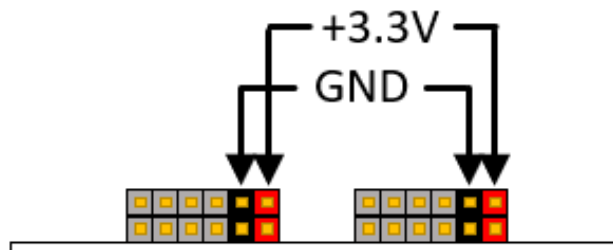


Figure 7.10 – Oscilloscope board power connections

The first step in verifying that there's a proper power connection is to ensure connectivity exists between the GND pins. Connect the black probe on your multimeter to the GND test point on the circuit board. Set the meter to the lowest resistance measurement range, which is typically 200 Ohms. Touch the red probe to each of the four ground pins on the Pmod connectors and verify that the meter drops to a very low resistance reading (less than 1 Ohm).

Next, test each of the +3.3 V pins with the red probe. The resistance to ground is hard to predict, but it should be somewhere in the range of a hundred to a few thousand Ohms. If the resistance is very small – under 50 Ohms – there is probably an unintended short in the circuit. If the resistance is very large – hundreds of kOhms or more – there may be unintended open connections between the power input and circuit components. If any problems are observed, a focused visual inspection of the problem area may identify the issue. If this is not successful, probing with the multimeter at intermediate points in the circuit may help isolate any problems.

With any problems identified during the electrical check resolved, the circuit is ready to be connected to the Arty board. Initial power application will be discussed in *Chapter 8, Bringing Up the Board for the First Time*.

Summary

This chapter introduced the processes and techniques involved in assembling high-performance digital circuits using surface-mount and through-hole electronic components. A recommended set of tools was identified, including a soldering station, a magnifier or microscope, and tweezers for handling tiny parts. Procedures for solder stenciling and preparing parts for assembly were presented in a step-by-step manner applicable to a wide variety of projects. After soldering, the steps involved in cleaning the board, performing a thorough inspection, and implementing any needed repairs were introduced.

Having completed this chapter, you should understand the tools and procedures required for solder stenciling and the steps involved in preparation for circuit board assembly. You learned how to solder surface-mount and through-hole components to the circuit board by hand and by using a reflow system, as well as how to clean the assembled board and thoroughly inspect it, including checking for electrical short circuits.

In the next chapter, we will power up the completed circuit board for the first time, verify that all the subsystems operate properly, and prepare to implement the remaining system functionality within the FPGA logic and as firmware running on the MicroBlaze processor within the FPGA.

Section 3: Implementing and Testing Real-Time Firmware

With prototype hardware available, the process of implementing and testing firmware begins. Part 3 takes us through the processes of firmware development, testing, and validation.

This part of the book comprises the following chapters:

- *Chapter 8, Bringing Up the Board for the First Time*
- *Chapter 9, The Firmware Development Process*
- *Chapter 10, Testing and Debugging the Embedded System*

8

Bringing Up the Board for the First Time

Having designed, constructed, cleaned, and inspected the printed circuit board, it is now time to apply power – in other words, perform the infamous smoke test. This chapter will lead you through the process of carefully providing first-time power to the board and checking basic circuit-level functionality. If you discover any problems, this chapter contains suggested approaches for fixing them. Once the board has passed these tests, we will continue to work on the FPGA logic, and will test the digital interface to the oscilloscope board.

After completing this chapter, you will have learned how to prepare the circuit for initial power application and how to test circuit components for proper operation. You will also understand how to identify and fix problems with the assembled circuit, and will have also checked out the digital interface to the circuit board.

We will cover the following topics in this chapter:

- Preparing for power-up
- Checking our circuit's basic functionality
- Adapting the circuit in case of problems
- Adding FPGA logic and checking I/O signals

Technical requirements

The files for this chapter are available at <https://github.com/PacktPublishing/Architecting-High-Performance-Embedded-Systems>.

To perform the checkout procedures on the digital oscilloscope circuit board, you will need a multimeter capable of measuring DC voltages. To examine the clock and data signals, you will need an oscilloscope with a bandwidth of at least 40 MHz.

Preparing for power-up

Chapter 7, Building High-Performance Digital Circuits, took us through the steps of constructing, cleaning, inspecting, and performing basic electrical checks on the digital oscilloscope circuit board. We are now ready to apply power to the board and perform testing to determine whether it is operating correctly.

Before applying power to the board, it is important to keep in mind that you need to exercise care when handling it and while it is operating. The integrated circuits on the board remain susceptible to damage from **electrostatic discharge (ESD)**, and it is easy to cause damage when you're probing parts of the circuit with metallic multimeter or oscilloscope probes. It is best to perform this work in an ESD-controlled environment, such as on the mat you use for soldering, and with a wrist strap in place.

If you must work with the board in a non-ESD-controlled environment, you should carefully handle the board by the edges and avoid touching the components on top of the board, the pins on the connectors, and any exposed pins or traces on the bottom of the board.

Supplying power to the board

The board requires +3.3 VDC as its input power source. If you have a standalone power supply that provides this voltage, you may want to use it for the initial power application.

If you do not have a separate power supply, you can use the Arty board to provide power. Our successful electrical check of the board at the end of *Chapter 7, Building High-Performance Digital Circuits*, indicated that there is substantial resistance between the +3.3 V power pins and board ground, which indicates the risk of shorting the Arty board power supply to ground is minimal.

The initial functionality checks will not require the Arty to perform any actions other than provide power to our board. If you are using a standalone power supply, set the voltage to +3.3 VDC and the current limit to 300 mA, if possible. Connect the +3.3 V and ground lines from the power supply to the board edge pins, as shown in *Figure 7.10* in the preceding chapter. You can choose any of the +3.3 V pins to provide power and use any of the GND pins to connect to the power supply ground. Double-check to ensure each of the connectors is on the correct pins.

If you are using the Arty board to provide power for the initial testing, ensure the USB cable is disconnected and that the Arty board is powered off before connecting the digital oscilloscope board to the Arty. Plug the oscilloscope board into the two center Pmod connectors on the Arty board. Make sure both rows of pins are inserted correctly and that the connectors are fully seated. The edge of the digital oscilloscope PCB should be flush against the edge of the Arty board.

The moment of truth has arrived: it is time for the **smoke test**. This term refers to the unfortunate situation where, sometimes, when a new circuit receives power for the first time, it produces smoke. This is obviously a very bad sign, and if it happens, you should remove power immediately, avoiding contact with the board if possible. If a board is producing smoke, some of the components on it will be very hot, and it is even possible for parts to explode, scattering pieces in a shrapnel-like manner. While such excitement is very unlikely with our 3.3 V circuit, it's always advisable to wear safety glasses during initial testing of a newly built electronic device.

Turn on the power to the board, either by enabling the power supply output (checking again to ensure the voltage is set to +3.3 V first) or by connecting power to the Arty board via the USB cable or through the board's power connector.

Observe the oscilloscope board for a moment. Nothing (such as smoke being emitted!) should be happening. If you are using an Arty to provide power, the red and green LEDs on the Arty should illuminate in their usual manner. If you are using a standalone power supply that displays the current being supplied, it should show a reading between 100 and 200 mA.

We are now ready to check the major subsystems of the circuitry, beginning with the on-board power supply voltages. These steps are the subject of the next section.

Checking our circuit's basic functionality

With power flowing to the board, we can start checking the DC behavior of the circuitry. This testing can be performed with a standard multimeter set to cover the range -4.0 to +3.3 V, which is typically the 20 V range.

Attach a clip lead to the multimeter's ground connection. Connect the ground clip to the GND test point on the digital oscilloscope circuit board.

Attach a probe-type lead to the multimeter's DC voltage input. This lead should come to a point, which will allow you to accurately contact small target locations on the PCB.

The following photo shows clip- and probe-type multimeter leads:



Figure 8.1 – Clip- and probe-type multimeter leads

We will be using the KiCad circuit schematic and PCB layout diagram to identify specific circuit points for testing with the multimeter. The schematic allows us to easily locate the features of the circuit we are interested in checking. The PCB diagram tells us where to find these points on the board.

When testing points on the circuit with the multimeter probe, it is important to be *very careful* to only touch the location you intend to examine. It is generally not a problem if you happen to touch the wrong place in the circuit with the probe. This is because the probe has a high impedance that has only a minimal effect on circuit behavior. However, problems arise if you happen to touch *two* circuit locations with the probe simultaneously. This can easily happen if you are trying to touch a single pin on an IC with closely spaced leads. Causing a short circuit in this manner can instantly destroy circuit components.

If you need to check the voltage on an IC pin, it is better to check some other component that is directly connected to the pin, such as a resistor, a capacitor, or a connector. With a bit of care, it is easy to touch one end of an SMT resistor or capacitor without any significant risk of creating a short to another circuit element.

If you find that you need to test a pin on an IC, it is best to check a pin at one end along the side of the IC, if possible. If you must check a pin that is surrounded by other pins, ensure you have good lighting, use magnification if necessary, and carefully slide the probe up so that it touches the pin. Ensure your hand does not slip while you glance at the meter. Then, carefully pull the probe away from the pin.

The first subject of our testing is the collection of power supply voltages generated on the board. This is the topic of the next section.

Testing the board power supplies

First, we will check the power supply voltages that are generated on the board: +2.5 V, -2.5 V, and +1.8 V. Let's begin with +1.8 V. The following diagram is the schematic for the +1.8 V power supply:

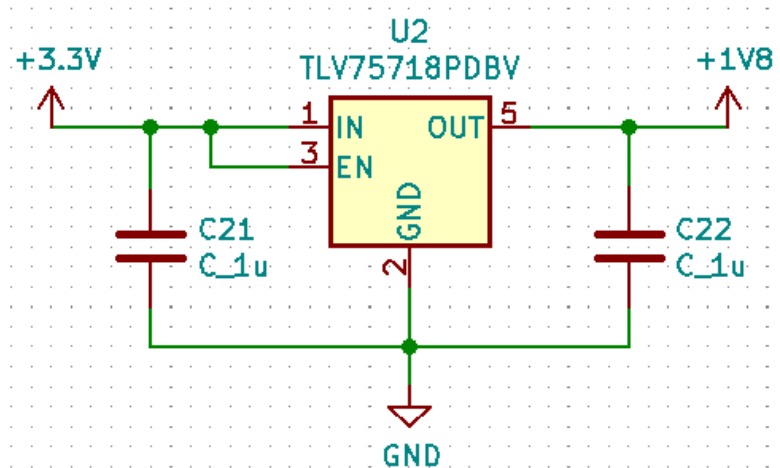


Figure 8.2 – +1.8 V power supply

The power supply output voltage, +1.8 V, is available at pin 5 of integrated circuit U2, as well as at the side of capacitor C22, which is connected to U2.

The following diagram shows the location of capacitor C22, along with some of the surrounding components, including U2:

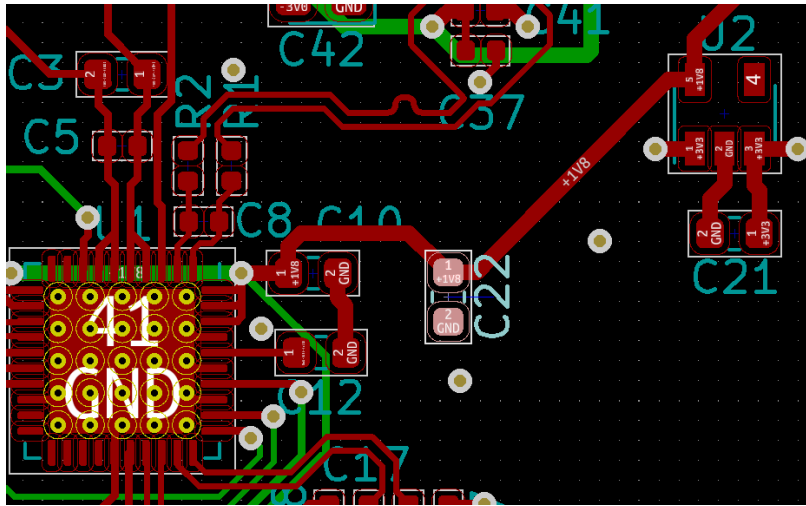


Figure 8.3 – Location of C22 on PCB

To check the +1.8 V supply voltage, perform the following steps:

1. With the digital oscilloscope board powered off and in a static-safe environment, connect the multimeter ground clip to the GND test point on the board.
2. Turn the multimeter on and set it to the 20 V DC range.
3. Apply power to the board.
4. Carefully touch the upper side of C22 with the multimeter probe (as shown in the preceding diagram) and, while holding the probe in place, observe the voltage measurement on the multimeter.

The voltage measurement should be within a few millivolts of 1.8 V. In my case, the multimeter provided a steady reading of 1.793 V.

Performing similar checks on the +2.5 V and -2.5 V power supplies, which can be accessed at the corresponding test point loops, yielded readings of +2.501 V and -2.506 V on my board, respectively. Your measurements may be slightly different, based on normal component variation. However, your voltages should be within a few millivolts of the target voltage.

If your testing produces similar readings – great! You are ready to move on to the next steps in checking out the board.

If any of the power supply readings differ substantially from the expected voltage – that is, by more than about 10 millivolts – you should stop what you're doing and attempt to resolve the problem. The following steps may be helpful when troubleshooting:

1. Perform another visual inspection, this time focusing on the components associated with the power supply in question. Attempt to identify whether any of the components are not making contact with solder pads. Check whether the power supply IC was installed with the correct orientation. See whether any solder bridges are present, or whether anything else is amiss with component installation.
2. Verify that the input to the power supply is at the expected voltage. For example, the +2.5 V and +1.8 V power supplies require +3.3 V as input to the voltage regulator IC. Ensure this voltage is getting to the chip.
3. Consider the possibility that incorrect components may have been installed at some locations. Make sure each resistor is the expected color – typically black – and that each capacitor is its expected color – typically a tan or brown color.
4. If you think it is possible that the incorrect values for some resistors or capacitors may have been installed, it might be necessary to remove the questionable components using hot air, and then solder new components that have been freshly removed from their packaging in place of them.

If none of these steps resolve a power supply voltage problem, there may be a cause other than an assembly error. If the current required by the components using the power supply exceeds the supply's capacity, the output voltage will droop. In this situation, it is necessary to return to the drawing board and reengineer the circuit. The circuit redesign must either provide a power supply with sufficient output current or modify the circuit design so that it uses components that draw less current than the capacity of the power supply.

If it turns out that changes need to be made to the circuitry to achieve acceptable performance, it may be possible to make some minor modifications directly to the PCB, without the need to incur the expense and delay associated with revising the circuit board. This type of modification will be covered later in this chapter.

If all the power supply voltages are correct, we are ready to proceed with checking out the remaining functional subsystems of the digital oscilloscope. We will continue by testing the analog amplifiers in the next section.

Testing the analog amplifiers

The oscilloscope input signal enters the board circuitry at the BNC connector, as shown in the following diagram:

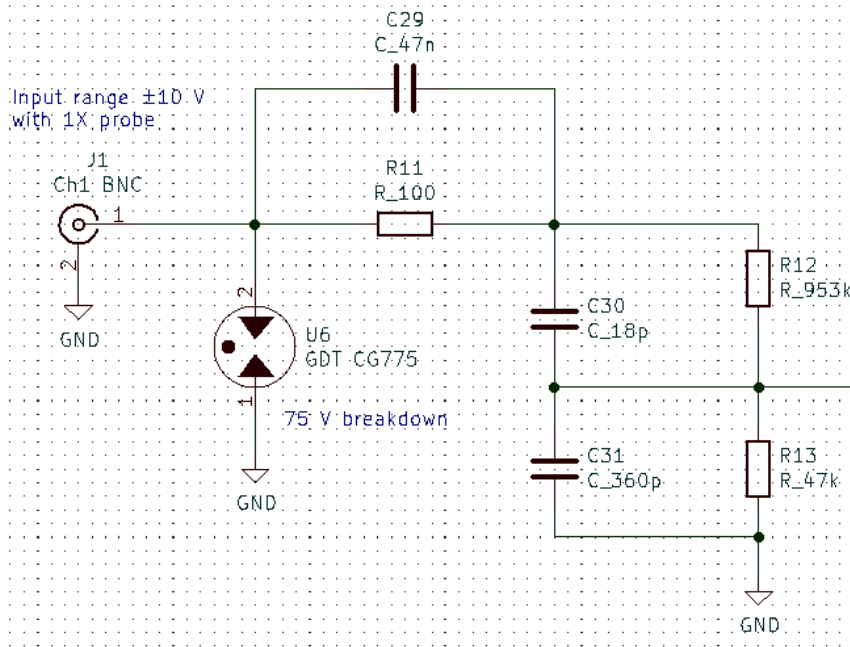


Figure 8.4 – Digital oscilloscope input circuitry

Our initial check will use a +2.5 V DC input signal to determine whether the circuitry is performing properly. When connected to a stable DC input signal, the capacitors and the gas discharge tube, labeled **GDT** in the preceding diagram, should have no influence on the voltage at different points in the circuit. The voltage at each point should remain constant.

The resistance of R11 is negligible in comparison to R12 and R13. The resistance of an oscilloscope probe when set to the 1X range (typically 100 to 300 Ohms) is also negligible relative to those two resistors.

With a constant +2.5 V input voltage, and ignoring the resistance of R11 and the scope probe, the expected voltage at the connection point between R12 and R13 can be determined as follows:

$$v = 2.5 \frac{47k}{953k + 47k} = 0.118 V$$

Perform a measurement of this circuit location using the procedures described earlier in this chapter. My reading was 0.121 V. If your measurement is not within a few millivolts of the expected value, attempt to identify and fix the problem.

If the voltage measurement is acceptable, the next step is to measure the output of **operational amplifier (op amp)** U7. U7 is configured as a unity-gain op amp, meaning the amplifier's output voltage is equal to its input voltage. The purpose of U7 is to provide a greater current-driving capability to the differential amplifier (U8) input than is available from the R12/R13 resistor divider network.

The following diagram shows the input and output connections of U7. Resistor R14 provides current limiting protection for U7 in case the input signal is connected to an out-of-range voltage. During normal operation, U7 provides a very high input impedance, which means there is negligible current through R14, and a corresponding negligible voltage drop across R14:

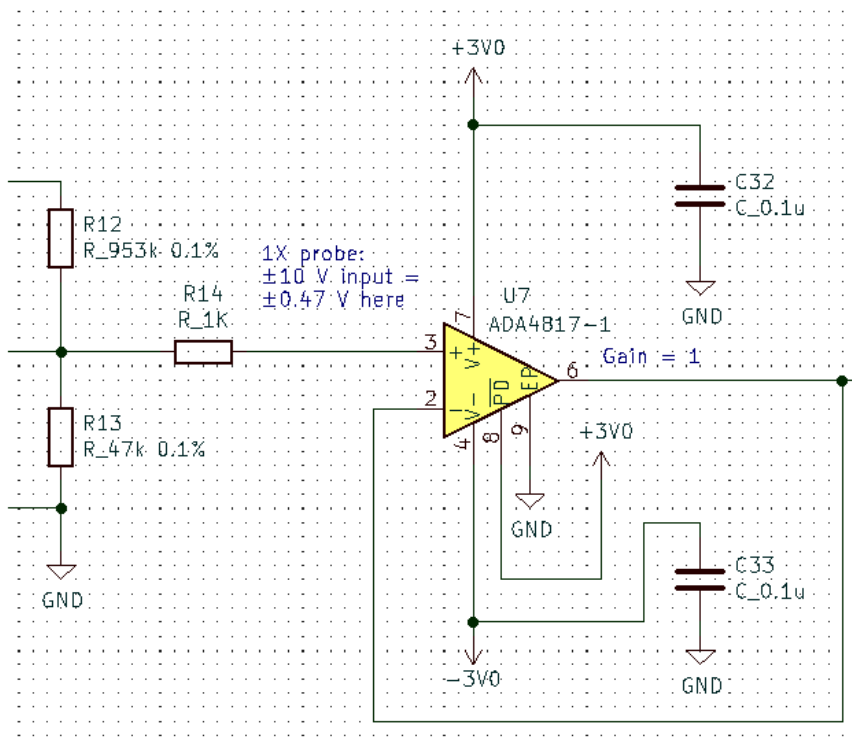


Figure 8.5 – Unity gain op amp

The signal we need to measure is on pin 6 of U7, which is not at the end of a row of pins. Pin 6 connects directly to pin 2, which is also not at the end of a row of pins. We can use the KiCad PCB layout to examine the network of connection points associated with these pins.

In the Pcbnew KiCad application, by performing a *Ctrl* + left-click on a circuit trace, the selected trace becomes highlighted and shows all of the points connected to it. The following diagram shows the trace connecting pins 2 and 6 of U7, as well as the connections to the input pins of U8. The collection of traces connecting multiple points in a circuit is referred to as a **net**:

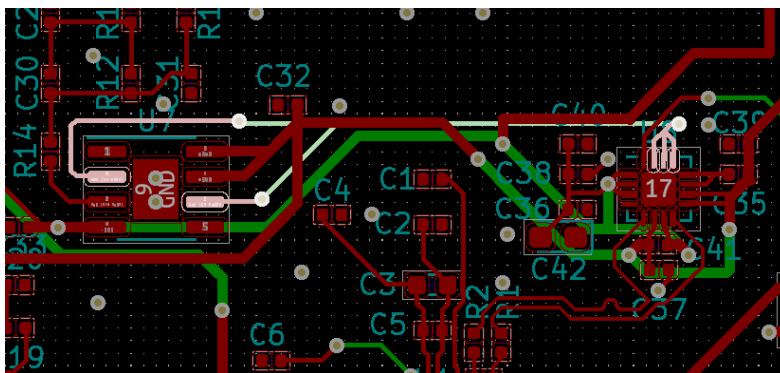


Figure 8.6 – Net connected to pins 2 and 6 of U7

Because the pins on U7 are fairly large, it may be reasonable to carefully measure the pin 6 voltage directly on the IC itself. It is also possible to probe one of the vias that connect the top and bottom traces on this net. In the preceding diagram, the vias are the circles at the connection points between the red top layer traces and the green bottom layer traces. The voltage on U7 pin 6 should match the voltage at the connection between R12 and R13, which is approximately 0.118V.

If the output voltage of U7 is correct, the next steps are to examine the outputs of the remaining amplifiers in the path to the ADC inputs: U8 and U9. Using an approach similar to the one we have used, you should calculate the expected output of each amplifier in response to the input voltage and then measure the output of the amplifier. U9 is a differential amplifier with an output voltage centered at the ADC input common mode voltage of 0.9V.

So far, our circuit check has verified that our amplifiers operate properly, up to the point of generating the differential signal that serves as the input to the ADC. The next step is to test the ADC itself.

Testing the ADC

The LTC2267-14 dual-channel ADC operates with a +1.8 V power supply. Although the device has two input channels, this design uses only one of them. This approach permits straightforward expansion of the design to support two oscilloscope input channels in the future. Using a **Serial Peripheral Interface (SPI)** configuration setting, the second ADC channel will be placed into *nap* mode to minimize power consumption.

The ADC has three major interfaces to other circuit components:

- **Analog input:** The analog input to the LTC2267-14 is a differential signal pair with an input configured for the range ± 1 V.
- **High-speed digital interface:** The digitized samples of the analog input are output to the FPGA using a two-lane LVDS interface. The clock used to drive these signals is provided by the FPGA.
- **SPI configuration port:** The SPI port on the ADC supports the configuration of various options, including output data format (offset binary or two's complement; 12-, 14-, or 16-bit output word size), per-channel *nap* mode selection, output driver current selection, and output test pattern control.

Having verified that the analog output of the differential amplifier is performing as expected, we assume for the time being that the ADC will receive that input. We will need to add some FPGA code to drive the high-speed digital interface from the ADC to the Arty board later in this chapter.

We can now begin to interact with the SPI configuration port on the ADC. To do this, a 6-pin ribbon cable is required to connect the SPI port on the Arty board to the SPI port on the oscilloscope board. Ensure that the pin 1 end of each cable is connected to the proper side of the connector. Both boards have a number 1 on them, indicating the location of pin 1. The type of cable required for the SPI connection is shown in the following figure:



Figure 8.7 – SPI connection cable

To use SPI to communicate with the ADC, we will continue developing the application we started in the *Kicking off the oscilloscope FPGA project* section of *Chapter 5, Implementing Systems with FPGAs*.

The first step working with the SPI connection is ensuring it is operating at a suitable clock speed. From the LTC2267 data sheet (available at <https://www.analog.com/media/en/technical-documentation/data-sheets/22687614fa.pdf>), we can see that in write mode, the SPI supports a clock period of 40 ns (25 MHz), while in readback mode, it supports a minimum clock period of 250 ns (4 MHz). Referring to the block diagram for our FPGA design, the `ext_spi_clk` input to the **AXI Quad SPI** block is driven by the 166.66667 MHz output of the **Clocking Wizard** block.

Before we can begin to use the SPI, we must perform the following steps to adjust the clock speed of the FPGA SPI and to fix separate a problem with SPI pin assignment:

1. Double-click the **AXI Quad SPI** block and select the **IP Configuration** tab.
2. The **Frequency Ratio** value of 16 indicates the `ext_spi_clk` input is divided by 16, producing a SPI clock rate of 10.4 MHz. This is too high for ADC readback mode. Change the multiplier in the box to the right of **Frequency Ratio** (to the right of X) from 1 to 10. This will reduce the SPI clock speed to 1.04 MHz.
3. Click **OK**.
4. Save the block diagram.
5. I experienced a problem with the SPI pin assignment. The SPI SS signal was assigned to the incorrect pin (V17), which prevented the interface from working properly. To fix this problem, add the following line at the end of `arty.xdc` and save the file:

```
set_property PACKAGE_PIN C1 [get_ports spi_ss_io]
```

6. Generate the bitstream.
7. Export the hardware (**File | Export | Export Hardware...**).
8. Open the Vitis project you created in *Chapter 5, Implementing Systems with FPGAs*.
9. Update the Vitis project hardware with the new definition you just exported from Vivado.

The **Board Support Package (BSP)** software provided in the Vitis project contains a driver for the Arty SPI interface. We will use this driver to perform communication between the application software and the ADC over SPI.

The LTC2267-14 ADC provides read and write access to five 8-bit internal registers, numbered 0–4. Details on these registers are available in the LTC2267-14 data sheet.

Register 0 is used exclusively for performing a software reset, which is triggered by writing a 1 to register bit 7. This reset must be the first step in the configuration process.

Registers 1–4 contain various configuration settings. Each of these registers can be read or written to at any time. Our code will perform a software reset via register 0 and then write configuration data to each of the four remaining registers. Once each register has been written, it will be read back and the received value will be compared to the value that was just written. Any mismatches during this comparison will result in a failed return status from the configuration routine.

To configure the LTC2267-14 via SPI, perform the following steps:

1. Create a new source file in the Vitis software project named `spi.h`. Insert the following code into `spi.h`:

```
// SPI interface to LTC2267 ADC
// SPI clock is 166.66667 MHz / (16 * 10) = 1.042 MHz
// LTC2267 max SPI clock speed (readback) is 4.0 MHz

// Configure SPI interface; Return TRUE if successful
int InitSpi(void);

// Returns TRUE if the value was successfully written
// to and read back from the register at reg_addr
int SpiWriteAndVerify(u8 reg_addr, u8 value);

// Pass hard-coded configuration data to the ADC via
// SPI and return TRUE if successful
int ConfigureAdc(void);
```

2. Create a source file named `spi.c` and insert the following code:

```
#include <xspi.h>
#include "spi.h"

static XSpi SpiInstance;

// Configure SPI interface; Return TRUE if successful
```

```
int InitSpi(void) {
    int result;

    result = XSpi_Initialize(&SpiInstance,
        XPAR_SPI_0_DEVICE_ID);
    if (result != XST_SUCCESS)
        return FALSE;

    result = XSpi_SelfTest(&SpiInstance);
    if (result != XST_SUCCESS)
        return FALSE;

    result = XSpi_SetOptions(&SpiInstance,
        XSP_MASTER_OPTION | XSP_MANUAL_SSELECT_OPTION);
    if (result != XST_SUCCESS)
        return FALSE;

    result = XSpi_Start(&SpiInstance);
    if (result != XST_SUCCESS)
        return FALSE;

    XSpi_IntrGlobalDisable(&SpiInstance);

    return TRUE;
}
```

The `InitSpi` function initializes the `XSpi` driver, performs a self-test on the FPGA SPI hardware, and configures the interface to manually assert the SS signal. This SS mode is necessary for the interface to support the requirements of the ADC SPI. The final steps start the SPI and disable interrupts from the SPI device.

3. Add the following function to the file:

```
// Send one byte to, or read one byte from the ADC
// Valid values for cmd: 0x00 = write, 0x80 = read
static int do_transfer(u8 cmd, u8 reg_addr,
    u8 output_value, u8 *input_value) {
    u8 out_buf[2] = { cmd | reg_addr, output_value };
```

```

u8 in_buf[2] = { 0 };
const int buf_len = 2;
u32 select_mask = 1;

// Valid commands: 0x00 = write, 0x80 = read
int result = XSpi_SetSlaveSelect(&SpiInstance,
                                select_mask);

if (result == XST_SUCCESS) {
    result = XSpi_Transfer(&SpiInstance, out_buf,
                          in_buf, buf_len);
    *input_value = in_buf[1];
}

return (result == XST_SUCCESS) ? TRUE : FALSE;
}

```

The `do_transfer` function will transfer one byte into an ADC register or read one byte from a register, depending on the value of the `cmd` variable.

4. The following function will write a value to an ADC register, read the value back from the register, and return a status value that indicates TRUE if all the steps were successful and the value read matches the value that was written. Add the following code to the `spi.c` file:

```

// Returns TRUE if the value was successfully written
// to and read back from the register at reg_addr
int SpiWriteAndVerify(u8 reg_addr, u8 value) {
    const u8 write_cmd = 0;
    const u8 read_cmd = 0x80;
    u8 input_value;
    int result;

    switch (reg_addr) {
    case 0:
        // The only valid value for reg 0 is 0x80
        result = (value == 0x80) ? TRUE : FALSE;
    }
}

```

```
        if (result == TRUE)
            result = do_transfer(write_cmd, reg_addr,
                                value, &input_value);

        break;

    case 1:
    case 2:
    case 3:
    case 4: {
        result = do_transfer(write_cmd, reg_addr,
                            value, &input_value);

        if (result == TRUE) {
            result = do_transfer(read_cmd, reg_addr, 0,
                                &input_value);
            xil_printf("Value read back %02X\n",
                      input_value);

            if (value != input_value)
                result = FALSE;
        }

        break;
    }

    default:
        result = FALSE;
    }

    return result;
}
```

In the `SpiWriteAndVerify` function, register 0 is handled as a special case. For registers 1–4, the code transfers the value into the register, reads the register, and returns the result of the comparison.

5. The `ConfigureAdc` function, with descriptive comments removed, is listed here. This function writes all five ADC configuration registers with hardcoded values and returns a status value indicating whether all the operations were successful:

```
// Pass hard-coded configuration data to the ADC via
// SPI and return TRUE if successful
int ConfigureAdc(void) {
    const u8 reg0 = 0x80;
    const u8 reg1 = 0x28;
    const u8 reg2 = 0x00;
    const u8 reg3 = 0xB3;
    const u8 reg4 = 0x33;

    xil_printf("Register 0: Writing %02X\n", reg0);
    int result = SpiWriteAndVerify(0, reg0);

    if (result == TRUE) {
        xil_printf("Register 1: Writing %02X\n", reg1);
        result = SpiWriteAndVerify(1, reg1);
    }

    if (result == TRUE) {
        xil_printf("Register 2: Writing %02X\n", reg2);
        result = SpiWriteAndVerify(2, reg2);
    }

    if (result == TRUE) {
        xil_printf("Register 3: Writing %02X\n", reg3);
        result = SpiWriteAndVerify(3, reg3);
    }

    if (result == TRUE) {
        xil_printf("Register 4: Writing %02X\n", reg4);
```

```
        result = SpiWriteAndVerify(4, reg4);  
    }  
  
    return result;  
}
```

6. In the `main.c` file, add the following line after the other `#include` statements:

```
#include "spi.h"
```

7. In the `main.c` file, at the very beginning of the `main` function, add the following code:

```
if (InitSpi() == TRUE) {  
    xil_printf("InitSpi success\n");  
  
    if (ConfigureAdc() == TRUE)  
        xil_printf("ConfigureAdc success\n");  
    else  
        xil_printf("ConfigureAdc failed\n");  
} else  
    xil_printf("InitSpi failed\n");
```

8. Save all the files you have edited. Ensure the `spi.h` and `spi.c` files appear in the Vitis Explorer window inside the `src` folder.
9. Press `Ctrl + B` to build the application. Ensure there are no errors.
10. Start the debugger and run the application.
11. If the SPI is operating as expected, you should see the following output:

```
InitSpi success  
Register 0: Writing 80  
Register 1: Writing 28  
Value read back 28  
Register 2: Writing 00  
Value read back 00  
Register 3: Writing B3  
Value read back B3  
Register 4: Writing 33
```

```
Value read back 33
```

```
ConfigureAdc success
```

Successfully completing this test indicates that the ADC integrated circuit is powered up and that the SPI interface connecting it to the Arty is working properly.

If you experience issues while checking out a new circuit design that can't be resolved by fixing errors that were introduced during assembly, it may turn out there are higher-level problems with the circuit's design. If the circuit requires fixes involving modifications to component connectivity, it may be possible to make some changes directly on the board and avoid the need for an immediate PCB revision. These techniques are the subject of the next section.

Adapting the circuit in case of problems

In *Chapter 7, Building High-Performance Digital Circuits*, we discussed various techniques for repairing problems resulting from improper assembly of the circuit board. The base assumption behind those procedures was that the circuit design was correct, and that any issues that arose were related to the assembly process.

You may reach a point where you identify one or more problems with the design of the circuit itself during testing. Once a design problem has been identified, it might be straightforward to revisit the circuit schematic and make the necessary corrections. The immediate problem, though, is that the PCB you are working with cannot be fixed as easily. Ordering a revised board will cost money and take time. It may be helpful to explore the possibility of modifying the PCB in order to implement immediate design changes.

Depending on the specific problem, it may be possible to make some modifications to the circuit board that will enable continued checkout of the remaining board capabilities. The modifications we will discuss in this section should only be performed on boards that are being evaluated by system developers. In general, due to the fragility of these alterations, boards with these changes should not be released for use by end users of the device.

The types of modifications we will discuss are cutting PCB traces, installing solder jumpers and jumper wires, removing components, and adding components.

Cutting PCB traces

If you decide that a trace connecting two points on the circuit should not be there, you can cut through the copper layer to break the connection between those points.

Use a razor knife or hobby knife to make the cut. Apply moderate pressure and make several strokes across the trace. Make sure you do not cut any adjacent components or traces. Once you've finished the cut, use your multimeter to test the resistance between the two points to ensure the trace is fully severed. If the cut ends of the traces connect indirectly through other circuit components, the measured resistance may not be infinite, even if the trace has been fully cut.

Important note

Use caution when working with a sharp knife: When cutting PCB traces, ensure the PCB is on a solid surface and does not wobble. Be very careful not to cut yourself or anything else, such as your work surface. Make sure your first aid kit is handy in case of an accident.

If cutting the trace was the only modification needed, you can now get back to testing the circuit with the trace no longer providing connectivity between its endpoints.

Typically, after cutting a trace, you will need to make connections between other points on the circuit board to correct the design error. This is the subject of the next section.

Installing solder jumpers and jumper wires

If you need to make a connection between two points on the PCB that are not currently connected, the first step is to examine the board and identify the available points where each trace or device pin is directly accessible.

If the points you need to connect are very close together, perhaps two adjacent pins on an IC or two resistors located very close to each other, you may be able to connect the points using a solder jumper. A **solder jumper** is just a lump of solder large enough to span the distance between the two points being connected. A solder jumper is the same thing as a solder bridge, as discussed earlier, except the solder jumper is intentional rather than undesired.

As always, use a sufficient amount of flux to ensure good flow and adhesion of the solder to the metal of the pads and components. A nice thing about solder jumpers is that if you decide you don't want the jumper at a later time, it is easy to remove using a soldering iron, flux, and solder wick.

If the distance to be bridged is greater than a reasonably sized solder jumper can connect, you will need to use a piece of wire to make the connection. If the jumper wire is a connection between points with no exposed pads or components between them, it may be reasonable to use wire that has no insulation. If the wire must cross metal pads and wind around components, the use of insulated wire may be mandatory.

Use a wire thickness that is appropriate for the connection endpoints and the current flow requirements. Jumper wire that attaches to the smallest SMT components should be of suitable fineness for connecting to such small points. If the wire is too thick, it will be difficult to attach to the desired location without coming into contact with other parts of the circuit.

The following photo shows a jumper wire, constructed from a bit of wire clipped from a through-hole resistor, connected between pins 3 and 4 of U4 on an earlier revision of the digital oscilloscope PCB:

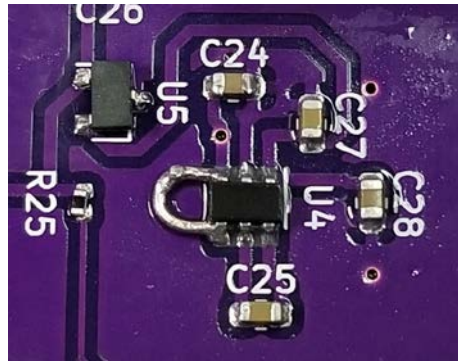


Figure 8.8 – Jumper wire connected across IC pins

If at all possible, use the location of a through-hole component, such as a board edge connector, as the connecting point for jumper wires. It is relatively easy to solder a wire to these parts where they pass through the PCB.

The trickiest type of jumper wire connection may be soldering the wire directly to a PCB trace. To do this, first scrape a sufficient amount of solder mask from the trace to expose an area large enough to make a good solder connection. Using flux, solder the jumper wire directly to the trace.

You can expect the jumper wire connections that are made using the methods described in this section to be quite fragile. Handle the board carefully to avoid breaking the wires free from their connections or causing more serious problems, such as lifting components or traces from the PCB surface.

Removing components

If you suspect an IC is drawing more current than anticipated, and thereby causing an associated power supply voltage to droop, a quick way to test for this is to remove the component from the board and check the supply voltage again. If the supply voltage is now at the expected value, you have your answer.

It may also be the case that a component is interfering with the operation of some other part of the circuit in other ways. By selectively removing resistors, capacitors, and ICs, you may be able to continue checking out portions of the circuit other than the area you have discovered is not operating correctly.

Adding components

If you discover the circuit is missing a necessary component, it may be possible to install the missing part so that you can continue checking out the circuit. If the missing part is a resistor or capacitor, it will likely be easier to install a through-hole component than an SMT component.

During early revisions of the digital oscilloscope PCB, I discovered that a missing 2.7K Ohm pullup resistor was causing the SPI interface to not work. The requirement for this resistor is clearly identified in the LTC2267-14 data sheet, but I overlooked it.

To get the SPI interface working and to continue the checkout process for the other board functions, I soldered a through-hole resistor to the board connectors, as shown in the following photo:

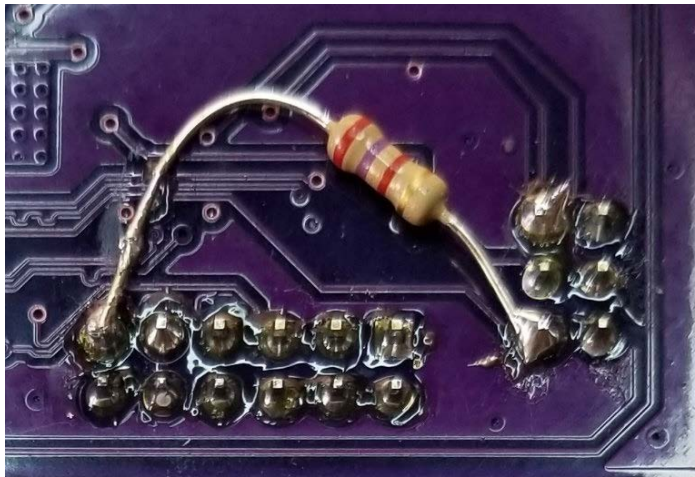


Figure 8.9 – Pullup resistor soldered to the PCB

Once any problems with the PCB have been at least temporarily resolved, the next step is to generate output signals from the FPGA so that we can drive the inputs to the oscilloscope board. This is the topic of the next section.

Adding FPGA logic and checking I/O signals

In this section, we will add the portion of the FPGA logic that generates signals that drive functions on the digital oscilloscope board. These signals include the 1 KHz calibration signal that is available on one of the board's test points and the ADC encoder clock, which drives the ADC.

Generating the ADC encoder clock and 1 KHz calibration signal

Continuing with the Vivado block diagram project for the digital oscilloscope we created in *Chapter 5, Implementing Systems with FPGAs*, and worked with previously in this chapter, we will now add logic to the FPGA design that will generate the ADC encoder clock and the 1 KHz output signal at the corresponding test point on the circuit board.

To minimize the bandwidth requirements for testing the board with an oscilloscope, we will temporarily reduce the ADC encoder clock frequency, which is intended to be 100 MHz. The ADC will accept a frequency as low as 5 MHz (per the LTC2267-14 data sheet) and operate the ADC at this slower rate. However, the lowest frequency we can readily generate from the **Clocking Wizard** is 10 MHz. We will set the ADC encoder clock frequency to 10 MHz for the time being.

Perform the following steps to make this frequency change and add the signals to the project:

1. Open the oscilloscope FPGA project in Vivado.
2. Open the project block diagram, then double-click the **Clocking Wizard** block.
3. Select the **Output Clocks** tab, then check the box next to **clk_out4**.
4. Set the frequency of **clk_out4** to 10 MHz and click **OK**.
5. Create a new design source file named `adc_interface.vhd`.
6. Replace the default content of the new file with the following code and save the file:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

library UNISIM;
use UNISIM.vcomponents.all;
```

```
entity adc_interface is
  port (
    adc_enc          : in std_logic;

    enc_p            : out std_logic;
    enc_n            : out std_logic;
    clk_1khz_out     : out std_logic
  );
end entity;

architecture Behavioral of adc_interface is
  signal clk_1khz      : std_logic;
begin

  process(adc_enc) is
    variable count      : integer := 0;
    constant clk_1khz_period : integer := 10 * 1000;
  begin
    if (rising_edge(adc_enc)) then
      count := count + 1;

      if (count >= (clk_1khz_period / 2)) then
        clk_1khz <= NOT clk_1khz;
        count := 0;
      end if;
    end if;
  end process;

  CAL_1KHZ_OBUF : OBUF
    generic map (IOSTANDARD => "LVCMOS33")
  port map (
    I => clk_1khz,
    O => clk_1khz_out
  );

  ADC_ENC_OBUFDS : OBUFDS
```

```

generic map (IOSTANDARD => "TMDS_33")
port map (
  O  => enc_p,
  OB => enc_n,
  I  => adc_enc
);
end Behavioral;

```

The preceding code receives the ADC encoder clock (`adc_enc`) that was produced by the Clocking Wizard (currently set at 10 MHz) and divides it to produce a 1.0 KHz signal. This 1.0 KHz signal is passed to the `clk_1khz_out` output using an output buffer (OBUF) driving a 3.3 V CMOS signal. The ADC encoder's output drives a differential signal pair (`enc_p`, `enc_n`) using the 3.3 V TMDS I/O standard (OBUFDS).

The Arty board's I/O signal configuration does not support the use of LVDS (the ADC's serial interface standard) on Pmod connectors, but it does support the **Transition-Minimized Differential Signaling (TMDS)** standard on those pins. TMDS is a high-speed serial data transmission standard similar in many ways to LVDS.

For our purposes, the main difference between LVDS and TMDS is that TMDS generates digital pulses by pulling the voltage down a few hundred millivolts from +3.3 V, while LVDS uses a lower common-mode voltage, about which it generates voltage pulses of a similar amplitude. To interface between LVDS and TMDS, we must accommodate the difference in common mode voltage between the two standards.

Our design uses DC blocking capacitors on each of the four differential signals leading to the Pmod connectors isolating the common mode voltage on each side of the capacitor. The Pmod connector side of these signals also has a 50 Ohm pullup resistor on each line, as required by the TMDS standard. This configuration enables bridging between the TMDS and LVDS I/O standards.

7. Add the following lines to the `arty.xdc` file:

```

# Pmod Header JC
set_property IOSTANDARD TMDS_33 [get_ports enc_p]
set_property PACKAGE_PIN U12 [get_ports enc_p]

set_property IOSTANDARD LVCMOS33 [get_ports clk_1khz_out]
set_property PACKAGE_PIN T13 [get_ports clk_1khz_out]

```

These statements include constraint information for the `enc_p` signal, but they do not mention the `enc_n` signal. This is because Vivado infers the need for the `enc_n` signal from the code in `adc_interface.vhd` and automatically assigns it to the correct pin with the appropriate properties.

8. Right-click on the block diagram's background and select **Add Module....**
9. Select `adc_interface` in the dialog that appears and click **OK**.
10. Connect the `clk_out4` output of the Clocking Wizard to the `adc_enc` input of the newly added **adc_interface_v1_0** block.
11. Right-click each of the outputs of the **adc_interface_v1_0** block and make them external.
12. Right-click each of the three newly added output ports and edit their properties. Remove `_0` from the end of each port name.
13. Generate the bitstream, export the hardware, and import the hardware configuration into the Vitis project.
14. Rebuild the project and run the application.

Once you've completed these steps, you can use an oscilloscope to examine the differential signal pairs involved in data transfer from the oscilloscope board to the Arty. To do this, you will need to refer to the PCB layout in KiCad and identify the appropriate connector pins or other circuit locations available for probing each of the following signal pairs:

- The **ENC_IN+** and **ENC_IN-** pins on J3 carry the ADC encoder clock, which we have currently set to 10 MHz. The following figure shows this differential pair on the oscilloscope:

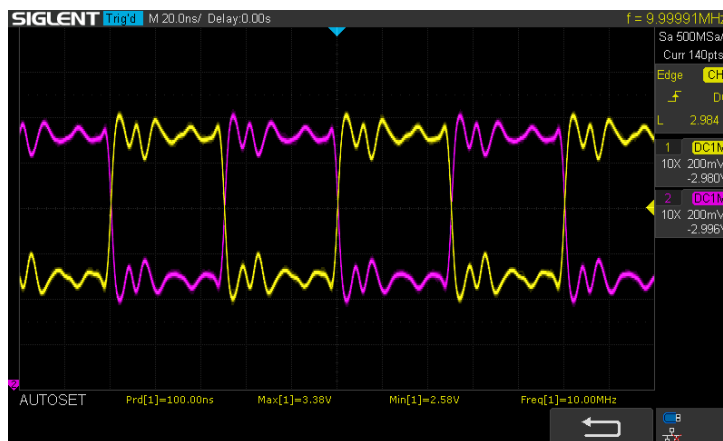


Figure 8.10 – ADC encoder clock signal

- The **DCO+** and **DCO-** differential pair contains the output data clock from the ADC. The frequency of this clock is the ADC encoder clock multiplied by 4, which results in a 40 MHz frequency, as shown in the following figure:

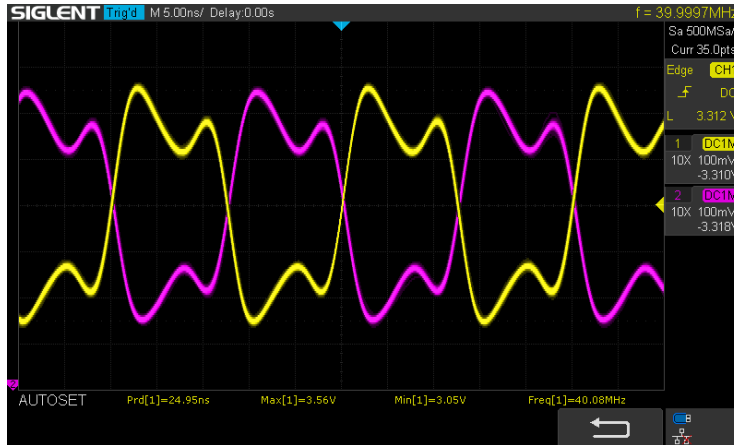


Figure 8.11 – DCO bit clock signal

- The values we loaded into ADC registers 3 and 4 ($\text{reg3} = 0xB3$ and $\text{reg4} = 0x33$) enable the ADC test pattern output mode and generate bit patterns on the differential **OUT1A** and **OUT1B** signal pairs. When connected to the oscilloscope, these signals should appear identical to the **DCO** outputs shown in the preceding diagram.
- The 1 KHz test point on the oscilloscope board should produce a square wave that oscillates between +2.5 V and -2.5 V at precisely 1 KHz.

If all these checks have produced satisfactory results, we have successfully demonstrated that almost all of the analog and digital subsystems on the digital oscilloscope board are operating properly. The only remaining untested area of functionality is the ADC conversion of analog inputs into digital outputs. We will work on this aspect of the oscilloscope in the next chapter.

Summary

This chapter led you through the process of carefully providing first-time power to the board and checking basic circuit-level functionality. After passing those tests, we added some FPGA code to generate the output signals that drive the oscilloscope board. We also discussed some ways we can modify and adapt the circuit if it turns out to not be functioning as intended.

Having completed this chapter, you now know how to prepare the circuit for initial power application and how to test circuit components and subsystems for proper operation. You have learned how to drive FPGA output signals and understand how to modify and adapt the circuit in case of design problems.

The next chapter will expand on the digital oscilloscope execution algorithm, including the remaining FPGA implementation, the firmware running on the MicroBlaze processor, and the software application running on the host computer.

9

The Firmware Development Process

Now that we have a functioning circuit board, it is time to flesh out some key portions of the **Field Programmable Gate Array (FPGA)** algorithm, including communication with the **analog-to-digital converter (ADC)**, and continue the development of the MicroBlaze processor firmware. When developing firmware, it is important to make use of appropriate tools to ensure that the source code is subjected to static analysis where possible, which can head off many errors that are otherwise difficult to debug. It is also important to implement a version control system to track the evolution of the code over the project life cycle. We will discuss the importance of developing a comprehensive, at least partially automated test suite to maintain code quality as changes are made. This chapter includes several recommendations for free and commercial tools that perform each of these functions.

After completing this chapter, you will have learned how to design efficient FPGA algorithms and develop embedded C code in a maintainable style. You will understand the basics of using static source code analysis with embedded system source code and will be familiar with the basics of Git version control. Finally, you will have learned the fundamentals of test-driven development as applied to embedded systems.

We will cover the following topics in this chapter:

- Designing and implementing the FPGA algorithm
- Coding style
- Statically analyzing source code
- Source code version control
- Test-driven development

Technical requirements

The files for this chapter are available at <https://github.com/PacktPublishing/Architecting-High-Performance-Embedded-Systems>.

Designing and implementing the FPGA algorithm

Up to this point, we have discussed only the details of the digital circuitry needed to receive an analog signal and perform the analog-to-digital conversion. We haven't really looked at what happens to an ADC sample after it has been captured. The next section will provide an overview of all of the functional elements of the digital oscilloscope system.

Digital oscilloscope system overview

Much of our work in the previous chapters has been focused on designing and constructing the hardware aspects of the digital oscilloscope add-on board that will plug into the Arty board. We will now examine the complete system at a high level.

The following figure provides a top-level view of the functional elements of the digital oscilloscope system:

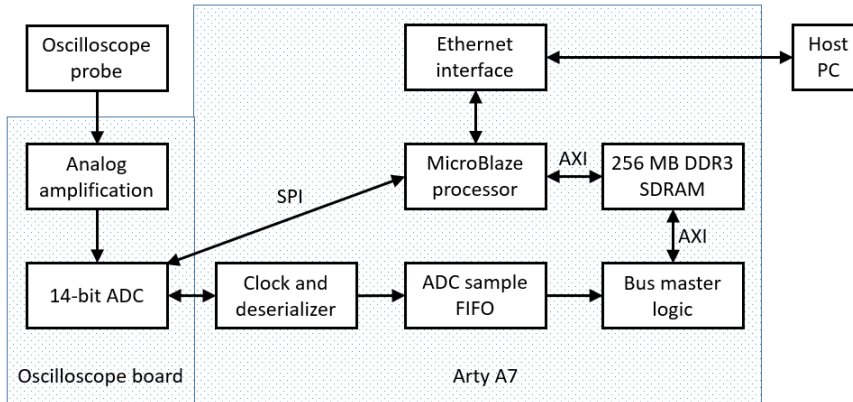


Figure 9.1 – Digital oscilloscope system diagram

Figure 9.1 shows the portions of the system that reside on the oscilloscope board, those on the Arty A7 board, and the connection of the Arty to a host PC. The host PC runs a software application that communicates with the MicroBlaze processor on the Arty board using the **Message Queuing and Telemetry Transport (MQTT)** protocol. MQTT provides reliable, message-based communication for applications that require a small code footprint, as in the case of our digital oscilloscope. MQTT is a popular protocol in IoT applications. To learn more about MQTT, visit the project website at <https://mqtt.org/>.

Under user control, the application running on the host PC formats and transmits the following information to the Arty board:

- *Trigger information*, such as the trigger voltage and edge (rising or falling). This data includes commands to start and stop monitoring incoming ADC samples for trigger conditions as well as information related to more complex trigger conditions such as the minimum high or low pulse width that must precede a trigger edge.
- *The number of samples to capture* once the trigger condition is satisfied. This includes an optional number of samples that precedes the trigger edge.
- *Configuration information*, such as test pattern mode selection and **bitslip** commands. The use of bitslip commands will be covered in the *Adding the deserializer* section later in this chapter.

In response to commands from a connected host application, the Arty board will gather and transmit sequences of ADC samples and related information such as the location of the trigger edge within the stream of samples. The host application receives each sample sequence from the Arty and renders a graphical display of the signal.

One aspect of the system that is worth emphasizing is that the ADC produces an absolutely enormous quantity of data. Each sample contains 16 bits of data consisting of a 14-bit sample plus two padding bits. The samples are produced at a 100 MHz rate. Many computer users consider a gigabit Ethernet card to be fast. In fact, Ethernet cards rarely operate at their maximum achievable speed for any sustained length of time. Our digital oscilloscope, on the other hand, with only one input channel, produces data continuously at 1.6 gigabits per second.

It would be a challenge for even the fastest modern microprocessor to ingest and process data at this rate. The FPGA architecture, in comparison, is ideal for this type of job. The naturally parallel operation of the FPGA logic gates enables the use of dedicated hardware to receive and process the incoming data stream.

Once commanded to start looking for a trigger condition, the FPGA logic will continuously monitor the ADC samples searching for the trigger. Because some number of pre-trigger samples must be captured, all ADC samples must be written to the 256 MB DDR3 memory while searching for the trigger event.

The MicroBlaze processor uses the same DDR3 memory to hold code and data for firmware running on the processor. The Vitis linker places these memory regions at the lower end of the entire DDR3 address space, which spans the hex addresses 80000000–8FFFFFFF in the default MicroBlaze memory map.

Each time you load the FPGA bitstream and the MicroBlaze firmware through Vitis, the **Xilinx Software Command-Line Tool (XSCT)** console window displays the list of memory sections as they are loaded and indicates the start and end addresses of each section. The processor stack memory segment, labeled `.stack`, is loaded into the highest memory addresses used by the application. DDR3 memory addresses above the end of the stack segment are available for use as storage for oscilloscope samples.

As development proceeds on the MicroBlaze firmware, it is critical that growth in memory consumption by the application code and data does not spill over into the memory used to store oscilloscope data samples. To minimize the likelihood of this problem, we can set aside a generous portion of memory for code and data. At the current state of development, the entire collection of application code and data, including the stack segment, consumes less than 200 KB of DDR3 memory. If we set aside 8 MB for future growth in code and data use by the MicroBlaze firmware, this leaves 248 MB of storage for oscilloscope samples.

248 MB of storage will hold 130,023,424 ADC samples, where each sample occupies 2 bytes. This corresponds to 1.3 seconds of continuous data samples at the 100 MHz rate.

When the FPGA logic detects a valid trigger condition, it will continue collecting ADC samples until the total number of samples requested has been stored to DDR3 memory. It will then stop writing to DDR3 and notify the MicroBlaze firmware that data collection has been completed. The MicroBlaze firmware can then read the data from DDR3 and transmit it over the network to the host application.

The following sections will describe the three Arty blocks in *Figure 9.1* that we have not worked with yet: the deserializer, the sample FIFO, and the bus master.

Adding the deserializer

Sample data collected by the ADC arrives at the Arty board edge connectors as two serial data streams (*IN1A* and *IN1B*) plus a clock (*DCO*). Each of these three signals consists of a differential pair. The *DCO* frequency is 400 MHz, which the ADC produces by multiplying the frequency of the ADC encoder clock, a 100 MHz signal provided by the Arty board, by 4. Although we have temporarily reduced the ADC encoder clock frequency to 10 MHz to make checkout and troubleshooting easier, we intend to raise it back to 100 MHz for normal oscilloscope operation. With this change, the frequency of the *DCO* signal is currently 40 MHz.

Each of the two sample data signals, *IN1A* and *IN1B*, consists of a series of data bits synchronized with *DCO*. New data bits are transferred on successive rising and falling edges of *DCO*, referred to as **double data rate (DDR)** format. By using DDR, data is transferred at 800 Mbit/s on each of the *IN1A* and *IN1B* signals, resulting in a total data rate of 1.6 Gb/s.

The ADC generates an additional clocking signal, called the frame clock, which designates the beginning of each 16-bit sample produced by the ADC. Unfortunately, the serial-to-parallel hardware provided by the Artix FPGA, called a **deserializer**, does not make use of a frame clock. This means each sample captured by the FPGA deserializer is not naturally aligned with the start of a sample. While it is possible the ADC samples will be aligned with the deserializer output samples, it is more likely the samples will need to be shifted by some number of bits to align ADC samples with deserializer samples.

By using the **Serial Peripheral Interface (SPI)** to specify a test pattern to be output from the ADC, we can examine the output of the deserializer and determine whether the samples are properly aligned. If the samples are misaligned, we can perform one or more bit-slip operations to achieve alignment.

The FPGA deserializer hardware provides an interface to request a bitslip, which shifts the deserializer output by one bit position relative to the incoming bitstream each time it is invoked. In general, it can take from zero to seven bitslips to align the deserializer output with the ADC sample stream. The sample alignment operation takes place under the control of the MicroBlaze processor each time the user commands the oscilloscope to begin looking for a trigger and should take no more than a few milliseconds. After the alignment process has been completed, the MicroBlaze firmware sends a SPI command to the ADC directing it to begin transferring digitized samples of its analog input over the serial interface.

We will now add the deserializer to the FPGA design. Continuing with the `oscilloscope-fpga` project containing the updates we made in *Chapter 8, Bringing Up the Board for the First Time*, the next step is to add interface signals to receive the data inputs into the FPGA from the digital oscilloscope board and feed them into a deserializer. The following steps will add these capabilities to the design:

1. Open the `oscilloscope-fpga` project in Vivado. The project must include the updates made in *Chapter 8, Bringing Up the Board for the First Time*.
2. Open the block design.
3. Right-click and select **Add IP...**
4. Type `selectio` in the search box and add a **SelectIO Interface Wizard** block to the diagram.
5. Double-click the new **SelectIO Interface Wizard** block on the diagram to open its configuration dialog.
6. In the **SelectIO Interface Wizard** block's **Data Bus Setup** tab, set **Data Rate** to `DDR`. Check the box next to **Serialization Factor** and set the serialization factor to 8. Set **External Data Width** to 2. Under **I/O Signaling**, set **Type** to `Differential` and **Standard** to `TMD5`. Verify that **Serialization Factor** is still 8 (change it back to 8 if it has been reset to a different value). Click **OK**.
7. On the **SelectIO Interface Wizard** block, click + next to `diff_clk_in` to display the clock input pins. Press `Ctrl + K` to create an external input. In the dialog that appears, set the port name to `dco_p`, **Type** to `Clock`, and **Frequency (MHz)** to 400. Connect this pin to the `clk_in_p` pin on the **SelectIO Interface Wizard** block. Repeat these steps to create a `dco_n` input connected to the `clk_in_n` pin.

8. Add two **Concat** blocks with two inputs each and connect their outputs to the `data_in_from_pins_p[1:0]` and `data_in_from_pins_n[1:0]` pins on the **SelectIO Interface Wizard** block. Create four external inputs of type Data connected to the **Concat** block inputs. Assign names to the ports as shown in the following figure:

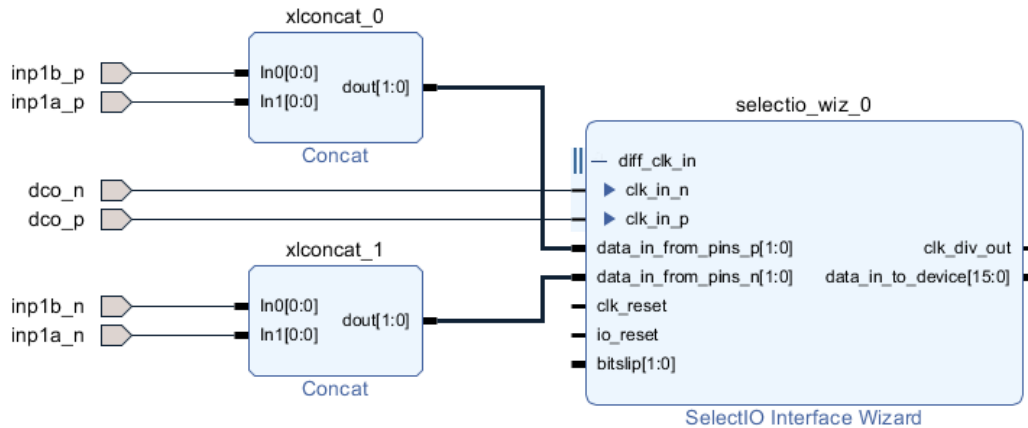


Figure 9.2 – Differential input signals

9. Add the following lines to the `arty.xdc` constraints file to define the pin connections for the input signals:

```
# Pmod Header JB
set_property PACKAGE_PIN E15 [get_ports in1a_p]
set_property IOSTANDARD TMDS_33 [get_ports in1a_p]
set_property PACKAGE_PIN D15 [get_ports dco_p]
set_property IOSTANDARD TMDS_33 [get_ports dco_p]
set_property PACKAGE_PIN J17 [get_ports in1b_p]
set_property IOSTANDARD TMDS_33 [get_ports in1b_p]
```


10. Edit `adc_interface.vhd` and add the incoming data and reset output signals required by the **SelectIO Interface Wizard** block to the port list:

```
entity adc_interface is
  port (
    adc_enc      : in std_logic;
    clk_div      : in std_logic;
    adc_data     : in std_logic_vector(15 downto 0);

    enc_p        : out std_logic;
    enc_n        : out std_logic;
    clk_1khz_out : out std_logic;
    clk_r        : out std_logic;
    io_r         : out std_logic
  );
end entity;
```

11. Add the following lines at the beginning of the architecture definition:

```
architecture Behavioral of adc_interface is
  signal clk_1khz      : std_logic;
begin
  process(adc_enc)
    variable clk_count : integer := 0;
  begin
    if rising_edge(adc_enc) then
      clk_r <= '0';
      io_r <= '0';

      if clk_count < 200 then
        if clk_count < 10 then
          null;
        elsif clk_count < 50 then
          clk_r <= '1';
        elsif clk_count < 100 then
          null;
        elsif clk_count < 150 then
```

```

        io_r <= '1';
    else
        null;
    end if;

    clk_count := clk_count + 1;
end if;
end if;
end process;

```

This code generates the reset signals required by the **SelectIO Interface Wizard** block.

12. Save the changes to `adc_interface.vhd`. You will be prompted to refresh the changed modules. Click where indicated to perform the refresh, then connect the inputs and outputs as shown in the following figure. Create a **Constant** block with a width of 2 and `Const Val` set to 00 and connect its output to the `bitslip[1:0]` input of the **SelectIO Interface Wizard** block. This is a temporary input to avoid warnings until we implement the `bitslip` interface:

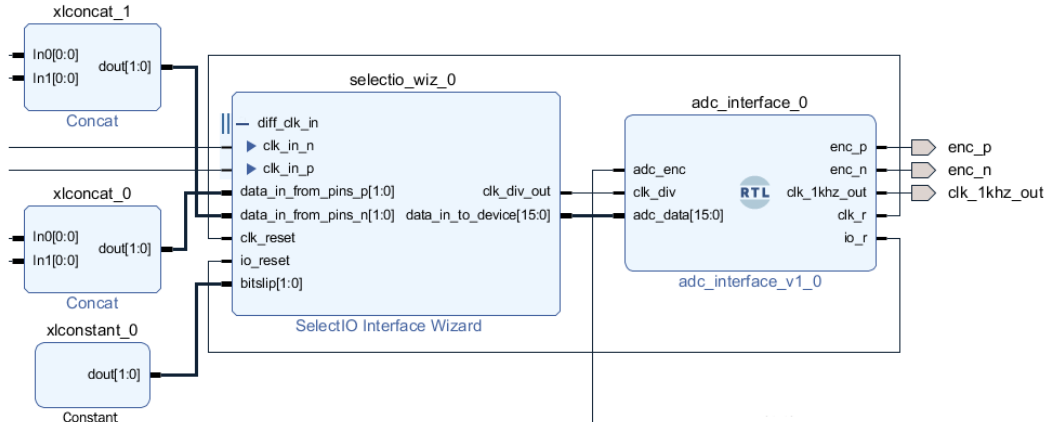


Figure 9.3 – SelectIO Interface Wizard connections

These steps have added a deserializer that outputs 16-bit ADC samples to the code in the `adc_interface_v1_0` block at a 100 MHz rate.

Because the transfer of samples to DDR3 memory cannot be assumed to keep up with this data rate at all times, the next step is to store the samples temporarily in a **first-in first-out (FIFO)** buffer. This is necessary because the data transfer to DDR3 uses the same bus as the MicroBlaze processor, which introduces delays during periods when multiple masters attempt to access the bus simultaneously. The use of the FIFO buffer allows samples to continue arriving without interruption even if access to DDR3 memory is momentarily unavailable.

Adding a FIFO buffer

Add a FIFO buffer with the following steps:

1. Right-click on the block diagram background and select **Add IP...**
2. Type **FIFO** in the search box. Select **FIFO Generator** from the list that appears and add it to the diagram.
3. Double-click the newly added **FIFO Generator** block. In the **Basic** tab, set **FIFO Implementation** to Independent Clocks Block RAM. In the **Native Ports** tab, select **First Word Fall Through**, set **Write Width** to 32, and **Write Depth** to 32768. Uncheck the box next to **Reset pin**. In the **Status Flags** tab, set **Programmable Full Type** to Single Programmable Full Threshold Constant and **Full Threshold Assert Value** to 8192. Click **OK**.
4. Add three FIFO outputs as shown at the bottom of the following port list to the `adc_interface` entity definition:

```
entity adc_interface is
  port (
    adc_enc          : in std_logic;
    clk_div          : in std_logic;
    adc_data         : in std_logic_vector(15 downto 0);

    enc_p            : out std_logic;
    enc_n            : out std_logic;
    clk_1khz_out     : out std_logic;
    clk_r            : out std_logic;
    io_r             : out std_logic;
    adc_fifo_wr_en   : out std_logic;
    adc_fifo_wr_ck   : out std_logic;
```

```

    adc_fifo_din      : out std_logic_vector(31 downto 0)
);
end entity;

```

5. Add the following code block to the architecture of `adc_interface`:

```

process(clk_div)
    variable adc_data_bits : std_logic_vector(15 downto 0);
    variable half : integer := 0;
    variable fifo_stage : std_logic_vector(31 downto 0);
    variable test_counter : integer := 0;

    constant half_offset : integer := 16;
    constant write_test_data : std_logic := '1';
begin
    -- The a channel contains odd bits (1,3,5,7,9,11,13)
    -- The b channel contains even bits (0,2,4,6,8,10,12)
    if falling_edge(clk_div) then
        adc_data_bits( 0) := adc_data(12);
        adc_data_bits( 1) := adc_data(13);
        adc_data_bits( 2) := adc_data(10);
        adc_data_bits( 3) := adc_data(11);
        adc_data_bits( 4) := adc_data( 8);
        adc_data_bits( 5) := adc_data( 9);
        adc_data_bits( 6) := adc_data( 6);
        adc_data_bits( 7) := adc_data( 7);
        adc_data_bits( 8) := adc_data( 4);
        adc_data_bits( 9) := adc_data( 5);
        adc_data_bits(10) := adc_data( 2);
        adc_data_bits(11) := adc_data( 3);
        adc_data_bits(12) := adc_data( 0);
        adc_data_bits(13) := adc_data( 1);
        adc_data_bits(14) := adc_data(14);
        adc_data_bits(15) := adc_data(15);
    end if;
end process;

```

```
-- Copy the ADC readings into the 32-bit FIFO buffer
adc_fifo_wr_en <= '1';

case half is
when 0=>
    if write_test_data = '1' then
        adc_fifo_din <= std_logic_vector(
            to_unsigned(test_counter, 32));
        test_counter := test_counter + 1;
    else
        adc_fifo_din <= fifo_stage;
    end if;
    fifo_stage((1*half_offset - 1)
        downto (0*half_offset)) := adc_data_bits;
    adc_fifo_wr_ck <= '0';
    half := 1;
when 1=>
    fifo_stage((2*half_offset - 1)
        downto (1*half_offset)) := adc_data_bits;
    adc_fifo_wr_ck <= '1';
    half := 0;
when others =>
    null;
end case;
end if;
end process;
```

This code receives the deserialized samples, places two sequential samples into a 32-bit word, and writes the word to the FIFO buffer. In test mode (when `write_test_data` is set to 1), it instead writes incrementing 32-bit numbers to the FIFO buffer. Test mode allows us to verify that all steps in data transfer from the FIFO buffer onward are working correctly and that no samples are being lost or duplicated.

We will next add an interface to the other side of the FIFO that reads data from the FIFO buffer and transfers it to DDR3 memory.

Adding the AXI bus interface

The Xilinx FPGA architecture and the MicroBlaze soft processor support a bus structure called the **Advanced eXtensible Interface (AXI)**. AXI provides a high-speed, parallel, multi-master communication interface intended for operation within an FPGA chip. Developers can define peripheral devices that interface to the MicroBlaze soft processor and other system components, such as DDR3 memory, using AXI. The primary function of the AXI in our design is to enable the transfer of ADC samples extracted from the FIFO buffer into DDR3 memory. Because the MicroBlaze processor is simultaneously accessing DDR3 to work with its code and data, we rely on the arbitration features of AXI to support the reliable transfer of ADC samples into DDR3.

Perform the following steps to add a peripheral to the AXI bus that supports writing data to DDR3 memory:

1. With the block diagram still open, on the Vivado **Tools** menu, select **Create and Package New IP...** Click **Next**, then select **Create AXI4 Peripheral**. Click **Next**. In the **Peripheral Details** page, enter `adc_bus_interface` as the name and click **Next**. Set **Interface Type** to **Full** and **Interface Mode** to **Master**. Click **Next**. Click **Finish**.
2. Right-click on the diagram background and select **Add IP...** Type `adc_bus` into the search box, then select `adc_bus_interface_v1.0` and add it to the diagram.
3. Click **Run Connection Automation**. Check the box for **All Automation**. Click `rd_clk` in the left column and change **Clock Source** to `/mig_7series_0/ui_clk (83 MHz)`. Click **OK**.
4. Right-click the `adc_bus_interface_0` block and select **Edit in IP Packager**. This will open a new copy of Vivado.
5. In the newly opened Vivado, expand **Design Sources** and open the master file (the file with `M00` in the filename).
6. Change `C_M_TARGET_SLAVE_BASE_ADDR` to `x"80000000"` and set `C_M_AXI_BURST_LEN` to `256`. Remove `INIT_AXI_TXN`, `TXN_DONE` and `ERROR` from the port list and add them as signal definitions in the `architecture` section. Save the file.
7. Edit the `adc_bus_interface_v1_0.vhd` file. Delete all references to the three signals you removed as ports in the previous step. Save the file.

8. Click the **Package IP – adc_bus_interface** tab. Under **Packaging Steps**, go from top to bottom and click each item that does not have a green checkmark and then click on any action in the yellow bar to incorporate corresponding changes. As the final step, click **Review and Package** under **Packaging Steps**. Click the **Re-Package IP** button, then when prompted, click **Yes** to close the project.
9. Back in the original Vivado project, click **Show IP Status** (or **Report IP Status**, if that text appears instead) in the yellow bar. Click **Rerun** if necessary, then click **Upgrade Selected** at the bottom. When prompted, click **Generate** to generate output products.

These steps have added an AXI bus master component containing example code that writes a sequential set of numerical values to a range of memory. The component then reads the data back from memory and verifies that the values that were read match the values that were written.

Our application will require substantial modifications to this example code to read data from the FIFO buffer we created earlier in this chapter and write it to appropriate addresses in DDR3 memory. This component will also serve as the interface to the MicroBlaze processor for receiving trigger configuration data and commands to start and stop data collection.

We will not delve further into the FPGA code that implements the functionality of the digital oscilloscope. The code is necessarily extensive, and examining it in detail would not introduce any new fundamental concepts related to the development of high-performance embedded systems. Feel free to browse the code available at the book website referenced in the *Technical requirements* section earlier in this chapter for additional insight. You should now have sufficient background in Vivado and VHDL to understand the code that implements the remaining oscilloscope features.

One major piece of functionality remains to be included in the MicroBlaze firmware: the network communication protocol that enables communication between the Arty board and a host application running on a potentially distant computer across the network. The addition of this capability is the subject of the next section.

Adding the MQTT protocol

The initial version of the application we developed in the *Baseline Vivado project* section of *Chapter 5, Implementing Systems with FPGAs*, included a TCP/IP echo server that we demonstrated in a local network environment. We will build upon this code to add support for communication using the MQTT protocol.

MQTT is a communication protocol based on the publish-subscribe paradigm and is intended to support machine-to-machine communication. In the MQTT **publish-subscribe** communication system, one or more information publishers package data into messages and publish the messages under associated topics. A **topic** is a character string that identifies a message category. Subscribers identify the categories of interest to them by providing topic names to the broker. The **MQTT broker** is a centralized server application with which all publishers and subscribers communicate. All MQTT communication takes place through the broker. Subscribers and publishers do not interact with each other directly.

Figure 9.4 shows the use of MQTT in the implementation of our network-based digital oscilloscope:

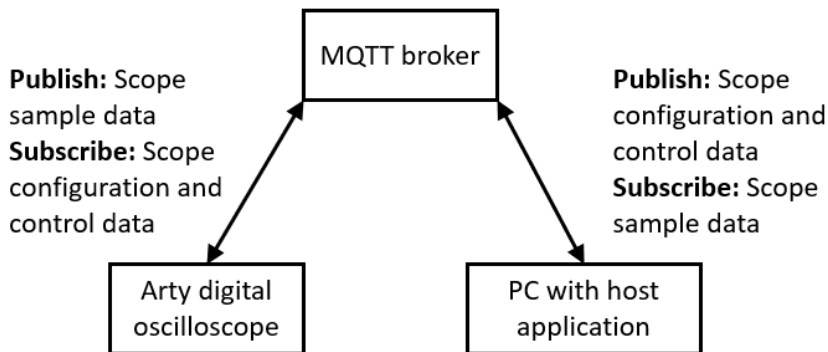


Figure 9.4 – Digital oscilloscope communication architecture

MQTT supports the simultaneous presence of multiple publishers and multiple subscribers, all connected to the same MQTT broker and publishing and subscribing to a common set of topics. For our system using the MQTT interface, it is possible to have multiple PC host applications receiving and displaying captured oscilloscope data from a single oscilloscope board simultaneously.

In the simple configuration of *Figure 9.4*, each PC host application has the ability to publish configuration and control commands to the oscilloscope as well. This obviously presents an opportunity for confusion if multiple users are trying to control the oscilloscope simultaneously. This type of communication challenge must be addressed to enable robust and well-managed operation of the device in a networked environment. The inclusion of this degree of distributed system management is certainly feasible, and is common in the context of IoT, but is beyond our scope in this chapter. Our immediate goal is to establish the ability for one user to interact with a digital oscilloscope in the manner represented in *Figure 9.4*. If multiple users happen to be working with the oscilloscope at the same time, it is up to them to coordinate their control operations.

The MQTT library we will be using operates within the TCP/IP networking context. This means the three system portions of *Figure 9.4* (the oscilloscope, the host PC, and the MQTT broker) can either be co-located on the same local area network, possibly using only a single PC, or all three can be at distant separations from each other, connected only by the internet.

Operating in its simplest configuration, MQTT does not provide any security against curious or malicious actors on the network. The default method of communication does not use any form of encryption or authentication of users. MQTT supports standard extensions available in TCP/IP to authenticate users and encrypt communications. For our initial iteration of MQTT communications, we will not address communication security. To avoid security concerns, the version of the code introduced in this section should only be run on a local network that is secured against external access. The addition of communication security is well documented by the MQTT project at <https://mqtt.org/>.

By default, the library source code generated by Vitis for this project includes an implementation of the MQTT protocol. Detailed information about the implementation of MQTT included in the FreeRTOS distribution is available at <https://www.freertos.org/mqtt>.

We will add the appropriate calls to the MQTT library functions to enable message publication and subscription. The initial implementation of MQTT functionality is a simple text string transfer that demonstrates the communication mechanism is working. We will not go into the details of implementing oscilloscope data transfer in this chapter. See the source code available at the book website for the details.

The following points summarize the key issues that must be addressed to add MQTT to the existing application code:

- **lightweight IP (lwIP) and FreeRTOS_IP:** The automatically generated echo server application uses the lwIP implementation of TCP/IP. lwIP is an older implementation of the TCP/IP protocol suite intended for use in resource-constrained embedded systems. More recently, FreeRTOS has been developing an implementation of TCP/IP optimized for the FreeRTOS environment. The MQTT example code we are using to implement our first iteration of network communication relies on the use of FreeRTOS_IP. Rather than converting the echo server code from lwIP to FreeRTOS_IP, it turns out to be simpler to port the MQTT example code to lwIP. lwIP is a robust and fully functional library so this does not create any problems. For clean slate development efforts, however, the use of FreeRTOS_IP should probably be preferred over lwIP due to its integration with FreeRTOS.
- **LWIP_DNS:** The echo server application uses the **Dynamic Host Configuration Protocol (DHCP)** to assign an IP address and other networking details for the Arty board. This works fine for operation on a local network, but if it is necessary for the application to operate over the internet, it is necessary for the TCP/IP stack to support **Domain Name System (DNS)** to look up a server's IP address given its name. For example, the publicly accessible MQTT broker located at `http://test.mosquitto.org/` has an IP address of `5.196.95.208`. The DNS service performs the translation from `test.mosquitto.org` to `5.196.95.208`. To enable the inclusion of DNS in lwIP, the preprocessor symbol `LWIP_DNS` must be defined when the lwIP code is compiled. You can set this up by editing the file named `Makefile` located within the project at `design_1_wrapper/microblaze_0/domain_microblaze_0/bsp`. Search for the text `EXTRA_COMPILER_FLAGS` and insert `-DLWIP_DNS` in the two locations where it appears. The surrounding text will appear as follows after you make this change:

```
EXTRA_COMPILER_FLAGS=-DLWIP_DNS -g -ffunction-sections
```

We will next discuss the calls to the MQTT API used in this example.

In the `main.c` file of the echo server application, the `main()` function starts up a thread named `main_thr`, which initializes the lwIP library and creates a thread named `NW_THRD`, short for network thread. The network thread configures the Arty Ethernet interface, starts a thread to receive incoming packets (which is required by lwIP), and issues a DHCP request. The main thread checks every 0.5 seconds to see whether the DHCP operation has completed. Upon successful completion of the DHCP operation, the main thread starts another thread to run the echo application, which is named `echo`.

We will leave the echo server capability in place and add the MQTT capability to the application. We will start the MQTT thread immediately after the main thread starts the echo server. The code to do this consists of a call to the function to start the MQTT demo:

```
vStartSimpleMQTTDemo();
```

The `vStartSimpleMQTTDemo()` function starts a thread named `MQTTLWDemo` that performs the MQTT demo.

The complete code for the `vStartSimpleMQTTDemo()` function is as follows:

```
static void prvMQTTDemoTask(void * pvParameters)
{
    (void) pvParameters;

    int xMQTTSocket;
    const uint32_t ulMaxPublishCount = 5UL;

    for(;;)
    {
        xMQTTSocket = prvCreateTCPConnectionToBroker();
        prvCreateMQTTConnectionWithBroker(xMQTTSocket);
        prvMQTTSubscribeToTopic(xMQTTSocket);
        prvMQTTProcessIncomingPacket(xMQTTSocket);

        for(uint32_t ulPublishCount = 0;
            ulPublishCount < ulMaxPublishCount;
            ulPublishCount++)
        {
            prvMQTTPublishToTopic(xMQTTSocket);
            prvMQTTProcessIncomingPacket(xMQTTSocket);
            vTaskDelay(pdMS_TO_TICKS(
                mqttexampleKEEP_ALIVE_DELAY));

            prvMQTTKeepAlive(xMQTTSocket);

            prvMQTTProcessIncomingPacket(xMQTTSocket);
        }
    }
}
```

```

    prvMQTTUnsubscribeFromTopic (xMQTTSocket) ;
    prvMQTTProcessIncomingPacket (xMQTTSocket) ;
    prvMQTTDisconnect (xMQTTSocket) ;
    prvGracefulShutDown (xMQTTSocket) ;

    vTaskDelay (pdMS_TO_TICKS (
        mqttexampleDELAY_BETWEEN_DEMO_ITERATIONS) ) ;
}
}

```

This code is located in `mqtt_task.c`. It performs the following sequence of operations in an infinite loop:

1. A call to `prvCreateTCPConnectionToBroker()`, which is defined in `mqtt_task.c`. This function calls lwIP functions to create a TCP socket and establish a TCP connection to the broker defined by the `mqttexampleMQTT_BROKER_ENDPOINT` preprocessor symbol. The endpoint can be defined as either an IP address in text form, such as `192.168.1.177`, or a domain name such as `test.mosquitto.org`.
2. A call to `prvCreateMQTTConnectionWithBroker()` creates an MQTT connection to the broker. This function is defined in `mqtt_task.c`.
3. A call to `prvMQTTSubscribeToTopic()` subscribes to the `mqttclient/example/topic` topic name. Topic names are character strings that use a multi-level syntax where the levels are separated by the `/` character. Topic names do not exist on the broker until at least one client (subscriber or publisher) creates them. The following call to `prvMQTTProcessIncomingPacket()` receives the response to the subscription request from the broker.
4. The following loop performs five publish operations, each of which sends the message *Hello Light Weight MQTT World!* to the broker, which passes it along to any subscribers to the corresponding topic.
5. After completing the five publish operations, the call to `prvMQTTUnsubscribeFromTopic()` removes the topic subscription. The following statements disconnect from the MQTT broker and close the TCP socket. After a delay, execution returns to *step 1*, repeating the entire sequence in an infinite loop.

The `(void) pvParameters;` statement may be unfamiliar to some readers. The functions that implement FreeRTOS tasks must accept an argument of the form `void * pvParameters`. This allows any type of data to be passed to the task for its internal use. If you do not need to pass any data into a task, you can indicate to readers of the code that you do not intend to use the argument with the `(void) pvParameters;` statement. This statement causes the value of the argument to be read, but then does nothing with it.

To demonstrate this code, you will need to set up a broker on a local machine. An excellent open source MQTT broker is available from the Eclipse project. Download the appropriate distribution for Windows, Linux, or Mac at <http://mosquitto.org/download/>. After completing the installation, if you are using Windows, the following command will start the broker running at the default TCP port number 1883:

```
C:\>"C:\Program Files\mosquitto\mosquitto.exe"
```

The `mqtt_profile.h` file in the Vitis project contains a definition for the broker endpoint. Determine the IP address of the system running the MQTT broker, then configure the application to use this address with a line similar to the following. Replace the IP address shown here with the IP address of the system running your MQTT broker:

```
#define mqttprofileBROKER_ENDPOINT "192.168.1.177"
```

After rebuilding and downloading the code to the Arty, run the application and watch for messages in the Vitis serial terminal window indicating successful connection and message publication.

You can run an MQTT subscriber on your PC (or another system on your local network) and display the messages published by the Arty board. Execute a command similar to the following to subscribe and display the messages:

```
"C:\Program Files\mosquitto\mosquitto_sub" -h 192.168.1.177 -t  
mqttclient/example/topic
```

This example has established a minimal MQTT capability that can be used as a starting point for developing the full bidirectional communication capability required for the digital oscilloscope application.

In the next section, we will discuss the need to develop and employ a consistent set of rules for formatting firmware source code to help ensure that it performs as intended and is maintainable.

Coding style

The C and C++ programming languages have many powerful features but also present many opportunities for developers to insert unintended behaviors that can manifest later as severe bugs. By following a set of code styling rules, you can substantially increase the likelihood that your code will perform as intend and, perhaps even more importantly, the code will be easier to read and understand for future maintainers.

Of course, simply following a set of style guidelines does not guarantee your code will be bug-free. The consistent application of coding style rules is just one part of an effective firmware development process that includes thorough testing and rigorous version control.

The following sections list some fundamental coding style guidelines applicable to C and C++. Similar rules can be applied to other programming languages we use, such as VHDL and C#. When multiple developers are working on a project, all of them should follow the same style guidelines. This allows each developer to readily understand code written or modified by others.

Naming things

When selecting names for code artifacts such as source files, functions, data types, and variables, it is always best to choose a descriptive name. Lengthy names are fine, as long as they are accurate and unambiguous.

Function names should state what the function does as an active phrase. Variable names that indicate a Boolean (true/false) condition should state what it means to be true. Global variables should have a capitalization form that distinguishes them from local variables. For example, the use of `CamelCase` is suggested for global variables and `snake_case` for local variables.

These are some examples of descriptive variable names: `BatteryCharge`, `TimeSinceLastByteRcvd`, and `input_data_valid`. The last of these is a Boolean variable, where the true state indicates the input data is valid.

These are some examples of descriptive function names: `ComputeBatteryCharge`, `SetDisplayBrightness`, and `ReadTemperatureAdc`.

Comments in code

It is best to minimize the need for explanatory comments in source code. If you can write functions that are brief and perform a single logical operation, statements that are succinct and clear, and assign names to code artifacts that unambiguously indicate the type of information they contain or the functionality they implement, you will go a long way toward eliminating the need for comments to explain what the code does.

This does not mean the use of code comments should be eliminated altogether. In particular, there are often situations where it is necessary to interact with hardware devices in a manner that is not self-explanatory. Comments should be added to such code to clarify its function.

As any substantial code base undergoes bug fixes and the addition of new capabilities, it is distressingly common for the description of the code provided by the comments to deviate from the actual implementation. When this happens, the comments become a negative, providing inaccurate information about the code. To avoid this situation, developers must actively maintain the text of comments with the same intensity they apply to maintaining the code itself.

Avoid literal numeric values

It is best to avoid sprinkling literal numeric values (such as 210) throughout code. Instead, think of a meaningful name to give the number and create a constant to hold the value under that name.

Braces, indentation, and vertical spacing

C and C++ do not mandate any particular layout of source code, as long as the correct elements are separated by some form of whitespace. This means it is up to developers to arrange code in a format that provides as much visual information as possible about its logical structure for ease of understanding.

C and C++ encapsulate sections of code in blocks within functions and within statements such as `if/else` and `for` loops. There are a few different approaches to arranging the opening curly brace (`{`) character of these blocks. One common recommendation for the best place to put this character is by itself on the line following the function definition or statement type. The following code, from the example code in the *Adding the MQTT protocol* section earlier in this chapter, should make this organization clear:

```
static void prvMQTTDemoTask(void * pvParameters)
{
    (void) pvParameters;
```

```
int xMQTTSocket;  
const uint32_t ulMaxPublishCount = 5UL;  
  
for(;;)  
{  
    ...  
}  
}
```

Consistent indentation should be applied at each level of code blocks. The C examples here use four spaces as the additional indentation at each block level. There is no universal agreement as to whether the indentation should use tab characters or spaces. My preference is spaces, because then I know the code will appear consistently when opened in different editing applications.

Blank lines should be used to separate **code paragraphs** within a function. A code paragraph refers to a sequential series of statements that work toward a common purpose. The example code in the *Adding the MQTT protocol* section of this chapter contains several code paragraphs. Avoid inserting multiple blank lines in sequence. Excessive vertical whitespace limits the amount of code you can display onscreen.

Prioritize readability and correctness

The most important attribute of source code is its clarity to the reader. It does not matter if the code works correctly if you cannot understand what it does – such code is unmaintainable. You must be able to read and understand code before you can undertake to modify and improve it. All other concerns, such as execution efficiency and the minimization of resource consumption, must be treated as lower priorities.

If the code is clearly understandable, you can then determine whether it implements the developer's intent. As you make changes to fix bugs and enhance functionality, it is vital to continuously evaluate the clarity of the changes you are making.

After you complete a series of changes, perhaps in preparation for the release of a new version, it is important to review your work as a whole. Think of this as an *elegance pass*, in which you review all of the changes you have made to ensure the code is clear, readable, and consistent. Verify that any names you have created are accurate and unambiguous. Ensure vertical spacing, including separation into code paragraphs, is appropriate. Confirm that any comments that may have been affected by your changes have been updated. Examine indentation to verify each code block is at the correct horizontal location. Only after you have completed this review should you consider submitting the code for any further reviews and testing.

Avoid premature optimization

Modern optimizing programming language compilers are exceptionally good at what they do. This means that for the most part, developers should not expend energy trying to improve the local efficiency of their code with small tweaks they think will help.

For example, if you need to divide an integer value by 4, programmers will sometimes replace the division operator with a right shift by two bit positions. They do this because they know the processor can perform a right shift instruction in a much smaller number of clock cycles than an integer division instruction.

The problem with this logic is that the compiler already knows this, and if optimization is enabled, it will make this substitution for you and will make 10 other performance enhancements at the same time that you never even thought of.

Performing a right shift as a substitution for division can introduce a bug if the value being shifted happens to be negative. If division is what you need, your code should just say so.

This is not to say developers should give no thought to the performance implications of the algorithms used in the code. In particular, when choosing algorithms for such operations as searching for a value in an array of length N , the selection of an algorithm that requires $\log(N)$ operations, as opposed to one that requires N operations, can form a make-or-break distinction in the success of your product.

Avoid implementation-defined behavior

Many aspects of C and C++ are not fully standardized and are documented as either implementation-defined or undefined behaviors. Implementation-defined aspects of the language are left to the compiler developer to determine. Undefined language aspects are not addressed by the standard and a compiler is free to respond to the use of undefined features in any manner it chooses.

When writing code that is intended to be portable and maintainable, it is important to avoid implementation-defined and undefined behavior to the extent possible.

You may be surprised to learn that the number of bits in a byte is not defined by the C language. Most processors and compilers use the standard 8-bit byte, but you cannot assume this will always be true when transferring code to a different system.

Probably the most common implementation dependency that should be avoided is the sizes of the predefined data types, such as `int`, `short`, and `long`. For portability, it is best to `#include` the C `stdint.h` header file (or `cstdint` in C++) to define a set of integral data types of specified widths. This file defines types such as `uint8_t` as an unsigned 8-bit integer type and `int16_t` as a signed 16-bit integer. Signed and unsigned integer types of widths 8, 16, 32, and 64 bits are defined in this header file.

There are many other types of implementation-defined and undefined behavior in the C and C++ languages. The use of static source code analysis, discussed in the *Statically analyzing source code* section later in this chapter, can point out occurrences of these issues in your code even if your compiler accepts the code without complaint.

Avoid unconditional jumps

The use of the `goto` statement to transfer execution control should be avoided entirely. The use, or worse, overuse, of this statement leads to what is derisively termed spaghetti code, where control jumps from place to place without any sensible rationale.

It is generally possible to form cleanly structured code that performs a required sequence of operations without resorting to unconditional jumps. The same logic applies to the inappropriate use of `break` and `continue` statements within loops, when those statements are used to perform unconditional jumps.

Minimize the scope of identifiers

The scope of identifiers should be limited to the code that has a need to access them. This includes such language features as variable definitions, type definitions, and function definitions. When we speak of global variables here, we mean variables that are defined outside of any function.

When a function or global variable is defined in a source file (specifically, without including the `static` keyword), that function or variable is automatically global in scope, and is thereby accessible to all code that is compiled into the same application. This is true even if other application source files do not explicitly declare the global item.

To avoid placing variables and functions in the application's global address space, the `static` keyword can be used to limit their scope to the current source file.

The following code presents an example of a global variable and function that are limited in scope to the current file:

```
static int32_t BatteryCharge;

static void ComputeBatteryCharge()
{
    ...
}
```

Be aware that declaring a `static` variable within a function means the variable retains its value between calls to the function. A static variable within a function is limited in scope to the current function, just like non-static (also called automatic) variables defined within the function.

Indicate that constant things are constant

If a pointer argument to a function is used only to read the data referenced by the pointer, the argument should be declared as a pointer to `const`. This makes it clear to users of the function that the data pointed to will not be modified by the function. It also causes the compiler to generate an error message if the function's code attempts to modify the data.

For example, the following function signature indicates that the `list` array will not be modified by the function:

```
uint16_t BinarySearch(const uint16_t list[], uint16_t size,
                      uint16_t key);
```

The same logic extends to the definition of all variables. If a variable is assigned a value when it is created and the value never changes, it should be declared `const`.

Automated code formatters

Much of the work associated with placing curly braces and properly indenting statement blocks can be automated through the use of source code formatting software. Many modern code editors include a feature to perform automated formatting of source code. For example, in Vitis, you can select the code you wish to format and press *Ctrl + Shift + F*. If you don't like the rules the formatter is using, you can change them by selecting **Window/Preferences...** and then selecting the **Additional | C/C++ | Code Style** section and modifying the formatting rules.

While automated code formatting is helpful, it only alters the whitespace throughout your code. It is still up to you to use a proper approach when naming things, declaring items as `static` or `const`, and so forth.

In the next section, we will discuss static source code analysis, which provides a comprehensive capability to identify subtle issues within your source code without even running it.

Statically analyzing source code

As the name implies, static source code analysis examines the source code for a computer program and provides a report on issues it identifies in the code.

What is static code analysis?

A static source code analyzer is similar in some ways to a compiler for the same programming language. Both tools ingest source code for a program and process it under the rules of the associated programming language, which is C or C++ in the current discussion.

The difference between the two types of tools is that the compiler intends to generate executable code that implements the logic defined in legal source code. A source code analyzer, on the other hand, performs an extensive assessment of the code, generally far beyond that performed by a compiler, and analyzes the code for compliance with a lengthy list of rules.

The output of the source code analyzer is a set of messages indicating potential problems it discovered in the code. It is then up to the developer to examine the source code associated with each message to determine whether changes are warranted to bring the code into compliance with the analyzer's rules.

Static code analysis tools

The original version of *lint* was developed as a static C source code analyzer for the Unix operating system in 1978 to highlight issues associated with source code portability between differing processor architectures.

Today, a number of commercial *lint*-like tools are available that provide exceptional capabilities for detecting subtle and not-so-subtle issues in C and C++ code. Some examples of these tools include the following:

- LDRA rules (<https://ldra.com/automotive/products/ldrarules/>) is a standalone rule-based source code checker. It can enforce industry-standard rules such as MISRA as well as user-defined rules.
- PC-lint Plus (<https://www.gimpel.com/>) performs comprehensive static analysis of C and C++ source code. It can perform checks for compliance with industry standards such as MISRA.
- Clang-Tidy (<http://clang.llvm.org/extra/clang-tidy/>) is a C++ source code analysis tool that performs an extensive collection of checks and recommends fixes to resolve issues.
- RSM (<http://msquaredtechnologies.com/>) is a source code quality analysis tool that analyzes C, C++, and other languages. RSM measures software metrics such as the count of lines of code and code complexity.
- ECLAIR (<https://www.bugseng.com/>) is a static source code analyzer that performs rule-based analysis and can automatically generate code that implements test cases for the analyzed code.

Although each of these tools has its own features and learning curve, for the purpose of presenting examples, the remainder of this section will use PC-lint Plus.

Important note

The **Motor Industry Software Reliability Association (MISRA)** (<https://www.misra.org.uk>) is an automotive industry collaboration that promotes best practices in the development of electronic systems for use in road vehicles. MISRA publishes standards for the use of C and C++ in automotive electronic systems. Each of these standards contains a collection of rules, many of which can be enforced with error messages for noncompliance by static source code analysis tools.

The following sections list some recommendations for making effective use of source code analysis tools.

Using static code analysis effectively

Beginning to use a static source code analysis tool, especially if you are working with a substantial base of existing code, can be overwhelming if not carried out in a methodical manner. Just running a source code analyzer on even a moderate collection of source files can generate hundreds, or often thousands, of messages with different levels of severity. How do you know where to start?

Working with existing code

You will generally need to do some configuration work before you can perform the first analysis of your code. This generally consists of setting up the tool for your compiler and specifying the `#include` paths your compiler searches for library header files. You must also provide the tool with the definitions of preprocessor symbols you use during compilation.

Some of the source code analysis tools include support for at least partially automating this process. PC-lint Plus includes a Python configuration program that requires you to have Python as well as its `regex` and `pyyaml` modules installed. If you don't have these tools, installation is straightforward.

To perform the configuration, you must determine the directory location of the compiler executable. From the console message produced in Vitis during a compile, we see the compiler filename is `mb-gcc.exe`. A directory search under the Xilinx installation location turns up this file at `C:\Xilinx\Vitis\2020.1\gnu\microblaze\nt\bin`.

With Python installed and the path to the PC-lint Plus executables added to the Windows PATH variable, the next step is to determine the compiler family most appropriate for your compiler. The following command lists the compiler families supported by PC-lint Plus:

```
pclp_config.py --list-compilers
```

From the list that appears, we select `gcc` as the appropriate family for the Vitis compiler.

The following command will generate the configuration files needed for PC-lint Plus to work with the `gcc` compiler used by Vitis:

```
pclp_config.py --compiler=gcc --compiler-bin="C:\Xilinx\
Vitis\2020.1\gnu\microblaze\nt\bin\mb-gcc.exe" --config-output-
lint-file=co-gcc.lnt --config-output-header-file=co-gcc.h
--generate-compiler-config
```

This command generates two files: a C header file (`co-gcc.h`) containing a list of preprocessor definitions used by the compiler and a PC-lint Plus configuration file (`co-gcc.lnt`) containing a set of configuration settings that will match the source code analysis to the compiler and the target processor. We will place these files in the directory containing the source code for the Vitis application.

Next, we need to create a PC-lint Plus configuration file containing project-specific configuration information, such as additional include directories beyond the system library directories and option settings to restrict the level of output message severity. This file is listed in the following code block:

```
co-gcc.lnt // Include the compiler configuration

-max_threads=4 // Enable parallel processing

// Project #include file paths
-IC:/Projects/oscilloscope-software/design_1_wrapper/export/
design_1_wrapper/sw/design_1_wrapper/domain_microblaze_0/
bspinclude/include
-I"C:\Projects\oscilloscope-software\oscilloscope-software\src"
-I"C:\Projects\oscilloscope-software\oscilloscope-software\src\
standard\common\include"
-I"C:\Projects\oscilloscope-software\oscilloscope-software\src\
platform"
-I"C:\Projects\oscilloscope-software\oscilloscope-software\src\
standard\mgtt\include"
-IC:/Projects/oscilloscope-software/design_1_wrapper/export/
design_1_wrapper/sw/design_1_wrapper/domain_microblaze_0/
bspinclude/include

-wl // Enable errors only

+e900 // Display error count
```

Execute the following command to perform the analysis:

```
pclp64 pclp_config.lnt *.c
```

This command performs static analysis of all the C files in the current directory.

In the next section, we will deal with the potentially large number of messages resulting from this command.

Begin with only the most severe messages

The `-w1` option setting in the PC-lint Plus configuration file indicates all messages will be suppressed except the most severe *error* messages. This eliminates a large number of messages that would otherwise appear and allows us to focus on areas where the source code analyzer is unable to interpret the code.

In the case of the current Vitis application containing the echo server and MQTT capabilities, only one error message appears:

```
--- Module:      mqtt_task.c (C)
mqtt_task.c  401  error 115: struct/union not defined
                xBrokerAddress.sin_addr.s_addr = *(long *)
                (ulBrokerIPAddress->h_addr_list[0]);

                ^
mqtt_task.c  386  supplemental 891: forward declaration of
'struct hostent'

                struct hostent *ulBrokerIPAddress;
                ^
```

This message indicates an error at line 401 in `mqtt_task.c`. The analyzer is telling us `struct hostent` is undefined at this point in the code. We know this message does not reflect the true state of our code because it compiles and runs successfully.

The problem here is that the `LWIP_DNS` symbol has not been defined, as we saw is required in the *Adding the MQTT protocol* section earlier in this chapter. Adding the following line to the project PC-lint Plus configuration file will define the symbol and eliminate the message:

```
-DLWIP_DNS
```

This defines `LWIP_DNS` as a preprocessor symbol for the analyzer. This results in zero messages after running the analyzer again.

The next step is to enable messages of *warning* severity to be displayed, in addition to any *error* messages. Changing the `-w1` setting to `-w2` is all it takes to do this. This results in 90 total messages from the analyzer. PC-lint Plus has two additional severity levels: `-w3` (include *informational* messages) and `-w4` (display all messages). Running PC-lint Plus on the application code with `-w4` results in 2,440 total messages. Clearly, we would not want to start our analysis at that level.

Resolving analyzer output messages

There are two basic approaches for dealing with each message produced by a source code analyzer: fix the problem indicated by the message or suppress the message.

If a message is truly irrelevant, or if you conclude that the payoff from fixing the issue is not worth the time and effort fixing it would require, it is reasonable to suppress the message. However, this step should not be taken lightly. Take the time to understand the cause of the message and read the message description in the PC-lint Plus manual. You will sometimes learn a particular message is the result of some rarely mentioned nuance of the language that may cause severe problems under the right circumstances.

You can think of the time you use to understand the causes and solutions for source code analyzer messages as a master class in C and C++ programming. You will learn a great deal about ways to avoid problems in your code, and the code you write in the future will be better for it.

Common source code analyzer messages

This section lists some of the more common types of messages you will receive from source code analyzers and suggests some ways to fix the problems:

- *Loss of precision during assignment:* As an example, this occurs when assigning an `int32_t` value to an `int16_t` variable. To eliminate the message, if you truly intend to lose the upper 16 bits during assignment, perform a cast to the smaller size before performing the assignment: `int16_t shorter = (int16_t) longer;`
- *Loss of sign in promotion from int to unsigned int:* Assigning a signed value to an unsigned variable of the same size may cause a problem if the value assigned is negative. Use a cast if you are sure this is what you want to do.
- *Unused include files:* This one is easy to fix. Unnecessary `#include` files clutter the top section of source code files and imply complexity that does not exist. Delete these lines.

- *Ignoring function return values:* Many standard library and project-specific functions return a value that may or may not be important for further processing. Code that calls functions capable of returning error indications should always check for errors. Other functions may return a superfluous pointer to data. Source code analyzers typically allow suppression of this type of message for specific functions. For example, many developers choose to ignore the return value from the standard library `printf` function.
- *Could be declared as pointer to const:* This message indicates a pointer references data that is never modified through the pointer. The pointer should be declared as a pointer to `const` in this case.
- *Symbol not referenced:* This can occur when arguments passed to a function are not used in the function. As indicated in the *Adding the MQTT protocol* section earlier in this chapter, a statement similar to `(void) pvParameters;` will indicate to readers and to the source code analyzer that the argument is not intended to be used. This message also appears if variables or functions are declared but never used.
- *External could be made static:* This indicates a globally declared variable or function could be confined to file scope by use of the `static` keyword.
- *Declaration of symbol hides symbol:* C and C++ allow you to define local variables within functions that have the same names as global variables. This is generally a bad idea, and the offending variable names should be revised to eliminate the message.
- *Possible access of out-of-bounds pointer:* Modern source code analyzers are capable of simulating the execution of all paths through your code. If an execution path allows the array index to exceed the array boundary, you will experience a classic C/C++ bug involving access to unallocated memory. The ability to detect this type of error at the source code level makes source code analyzers worth their weight in gold.

In a typical code base, you will come across many occurrences of these and other messages. Most likely, you will find at least one occurrence of a true bug, which might have been very difficult and tedious to identify, track down, and fix using traditional debugging methods. By resolving these issues at the source code level, you can eliminate entire classes of problems.

In the next section, we will examine tools and processes for managing the version history of our source code files.

Source code version control

For any software project larger than a single-file program, it is critical to maintain a rigorously managed history of file versions. Several new and old software tools are available for performing version control, some of which are free and some offered commercially.

Rather than list the options available for these tools, we will focus on the popular version control system named Git. Git is the version control system used for the Linux operating system source code. Many online Git repositories such as GitHub (<https://github.com>) are available. Git is free and is available at <https://git-scm.com/downloads>.

Version control with Git

Vitis contains integrated capabilities for working with Git to version control your application source code. It also has the ability to work with remote repositories such as on GitHub or on an organization-provided Git server.

Git allows multiple developers to work on the same code base and enables each developer to bring their changes into the common repository.

Git is a distributed version control system, which means there is no concept of one user *checking out* a file to work on it. Each developer has a complete copy of the repository at all times and can update the local copy of the repository to incorporate changes from other developers at any time.

You can access the Git features in Vitis by right-clicking on the application project in the **Explorer** window in Vitis and selecting the **Team** option. Rather than delve into the details of Git in this chapter, consult the internet to locate tutorials for using Git in Eclipse. One example is at <https://dzone.com/articles/tutorial-git-with-eclipse>.

The next section will introduce the benefits of test-driven development for embedded systems.

Test-driven development

Test-driven development (TDD) is a philosophy and process for integrating comprehensive testing into the software development process at the earliest stage possible. The idea behind this approach is to begin with a set of fully tested individual components and continue testing as these components are integrated into a functional system. By using this methodology, the likelihood of significant bugs remaining in the code is drastically reduced.

To make the development process *test-driven*, you first write a test for a piece of functionality that does not yet exist in the system and then run the test. The test should fail, perhaps by not even compiling successfully if a function being called does not yet exist. You then implement the function the test is attempting to execute, which should allow the test to pass. The set of tests should verify as fully as possible that the implemented system code is performing correctly.

While fairly simple in concept, there are some challenges that must be addressed before TDD can be applied to a project, particularly if the project is an embedded system.

TDD applied to embedded systems

TDD has traditionally seen limited use in the development of firmware for embedded systems due to unique challenges in this environment, such as heavy dependence on the hardware features of embedded processors and peripherals.

If we can provide simulated, or **mock**, implementations of these hardware peculiarities, we can build and run a set of tests on the host computer. Assuming code compiled for the host computer performs the same as code compiled for the embedded processor, which is a pretty good assumption (especially if we use `gcc` as our host compiler), tests run on the host should validate the behavior of the code on the target processor. This approach avoids the need to download the code to the device and then figure out how to test behavior at the function level.

A testing framework for C named Ceedling (<http://www.throwtheswitch.org/ceedling>) simplifies and automates much of the otherwise tedious work that would be involved in configuring and running a test environment for embedded C code, including the generation of mock interfaces to embedded hardware.

Ceedling requires the installation of Ruby (<https://rubyinstaller.org/>) and Cygwin (<https://www.cygwin.com/install.html>), which provides the `gcc` compiler for host-based testing.

As with the initial configuration of a static source code analyzer, setting up a TDD capability involves some work and comes with a learning curve. With that said, the benefits of TDD in terms of early problem discovery are immense, and are especially helpful for completing a project on schedule. The use of a TDD process goes a long way toward avoiding the indeterminate-length troubleshooting and debugging sessions that are all too common in the development of complex embedded systems.

Summary

This chapter covered the implementation of a few of the significant remaining portions of the FPGA design, including the deserializer, the FIFO buffer, and the interface to the AXI bus. We covered the application of appropriate code style guidelines and discussed the use of static source code analysis as a powerful means of preventing many errors that are otherwise difficult to debug.

The chapter discussed the use of Git as a software project version control system. The benefits of TDD were discussed and the Ceedling TDD tool for C language projects was introduced.

Having completed this chapter, you understand the basics of designing FPGA algorithms and how to develop embedded C code in a maintainable and well-tested style. You are familiar with the basics of Git version control and understand the fundamental steps of TDD.

The next and final chapter will discuss best practices for performing thorough testing of the entire embedded device and will offer some effective approaches for debugging problems that are identified at this late stage in the development cycle.

10

Testing and Debugging the Embedded System

As the development of our embedded system nears completion, the time has arrived to conduct thorough testing in the context in which it will operate. This testing must address the entire expected range of environmental conditions and user inputs, including invalid inputs, to ensure proper operation under all conditions.

For each test, the system configuration and test execution procedures must be carefully recorded and any resulting behavioral anomalies noted in detail. If you don't have enough information to reliably repeat a test, it may be difficult or impossible to fix the underlying problem. The chapter concludes with a discussion of recommended debugging procedures and a summary of best practices for high-performance embedded system development.

After completing this chapter, you will understand how to efficiently and thoroughly test a complex embedded system. You will have learned appropriate procedures for running tests and recording test findings as well as techniques for effectively tracking down program bugs. You will understand how to develop a set of tests that evaluate all of the important system aspects and how to maintain a focus on the best practices of successful embedded system development.

We will cover the following topics in this chapter:

- Designing system-level tests
- Conducting tests and recording results
- Ensuring comprehensive test coverage
- Effective debugging techniques
- Summary of best practices for high-performance embedded system development

Technical requirements

Files for this chapter are available at <https://github.com/PacktPublishing/Architecting-High-Performance-Embedded-Systems>.

Designing system-level tests

At this point in the system development process, we will assume that our high-performance embedded device has been designed and constructed, and that an initial checkout of its basic functionality indicates everything appears to be working properly. It is time to subject the system prototype to a comprehensive set of tests to ensure it behaves as intended under all anticipated operating conditions, as well as in response to all forms of valid and invalid user input.

While this may seem straightforward, it is in fact a formidable challenge. As a simple example, consider a system that merely accepts as input a text string entered by the user. If the length of the string is not restricted, the universe of potential inputs is effectively infinite. It will never be possible to test all possible inputs the system might receive. Even for a simple system such as this, we must carefully decide what kinds of tests are needed and how much testing is enough.

For a system that has a sophisticated user interface and that simultaneously receives inputs from a variety of interfaces, the problem is compounded. Because it is impossible to test all possible inputs to any complex system, it is incumbent on the system developer to develop an adequate testing regime. The collection of tests must be executable within the time available in the development cycle and the tests must have a high probability of detecting any significant problems that remain in the system.

Because system testing takes place at the end of a development cycle, it is traditional for the testing phase to be severely constrained in terms of the time and resources available. In the face of these pressures, the tester must perform the tasks of designing and conducting tests as efficiently as possible. It is also important to ensure sufficient testing takes place, even if it results in extending the development period beyond the planned timeframe.

It does no one any good to rush an insufficiently tested product to market. Giving in to such pressures results in product recalls, a bad reputation for the product and the company, and potential legal consequences if a defective product causes harm to its users or others.

The following sections suggest some approaches for efficiently designing and executing tests, and documenting the results of those tests.

Requirements-driven testing

Although we do not have the space in this chapter to examine the entire requirements-driven system engineering process in depth, it will be helpful to briefly review the requirements generation process in the context of system testing. Whether or not your embedded system has a formal requirement specification document, you should be able to clearly state the things the system is supposed to do and the things it shouldn't do.

At the highest level, system requirements state the basic functionality the system implements and provide measurable threshold performance values against which the system can be evaluated. For example, if the device is powered by a rechargeable battery, a top-level requirement should specify how long the device must operate before the battery is exhausted.

A complete set of top-level requirements quantifies all of the basic features users expect from the system. Some examples of these requirements are listed here:

- Processing speed when performing specific tasks, in terms of the specific tasks and execution time constraints for each
- Sustained data transfer bandwidth under realistic operating conditions
- Screen resolution, brightness, and update rate
- The accuracy of measured inputs, such as temperature

Beyond the most obvious requirements that describe what the device is supposed to do lies a set of non-functional requirements. **Non-functional requirements** describe attributes the system must possess or other mandatory aspects not related to its behavior. Many of these requirements are obvious when stated, but until they are documented they exist only as unspoken assumptions. For example, *do not burst into flames* states an often-assumed requirement for devices containing rechargeable batteries.

Requirements can define invariant conditions the system must maintain regardless of the presence of valid or invalid inputs. A few examples of these are as follows:

- The operating system and application code do not crash.
- The system remains responsive to user inputs at all times.
- The system continues to operate properly while rejecting invalid input.

If your system lacks a complete definition of requirements, including those that are assumed, you may need to do a bit of brainstorming to produce a list suitable for use as input to the test process. System requirements provide the information you will need to define a set of relevant tests and then determine whether the system passes or fails those tests.

When developing system requirements, an analysis should be performed to verify that each requirement satisfies the following criteria:

- **Completeness:** The requirements cover all aspects of necessary system behaviors as well as all relevant non-functional conditions the system must satisfy.
- **Clarity:** Requirements must be defined in terms that all relevant parties understand and agree upon.
- **Testability:** Each requirement must be stated in terms that permit ready evaluation of the system's compliance. If you are unable to identify a way to test whether the system satisfies a particular requirement, the requirement should be rewritten in testable terms.

The following table contains a set of basic requirements for the network-based digital oscilloscope we have been working with as an example project. Each requirement has an associated identifier that provides a shorthand reference to the full requirement text:

Requirement ID	Description
R-1	The oscilloscope shall have a single input channel consisting of a BNC connector for use with standard oscilloscope probes.
R-2	The oscilloscope shall sample its input signal at a 100 MHz rate.
R-3	The oscilloscope shall support input signals over a range of ± 10 V when using a 1X oscilloscope probe.
R-4	The oscilloscope shall support input signals over a range of ± 70 V when using a 10X oscilloscope probe.
R-5	Oscilloscope samples shall contain 14 bits of resolution.
R-6	The oscilloscope shall collect and store up to 1.3 seconds of continuous measurements.
R-7	The oscilloscope shall transfer collected data to a host application over a TCP/IP-based network.
R-8	The oscilloscope shall support rising and falling edge-based triggering.
R-9	The oscilloscope shall support pulse width-based triggering.
R-10	The oscilloscope shall store a user-selected number of pre-trigger samples.
R-11	The oscilloscope shall store a user-selected number of post-trigger samples.
R-12	The oscilloscope shall receive and respond to user command information received over the network.
R-13	The oscilloscope shall tolerate normal handling of the signal input connection, including static electric discharge.
R-14	The oscilloscope shall operate over a temperature range of 0 to 70°C.
R-15	The oscilloscope shall operate over a relative humidity range of 8 to 80%.
R-16	The oscilloscope shall maintain a collection of status information and provide this data to the host application upon receiving a request.
R-17	The oscilloscope shall ignore invalid or out-of-range command data and set a status indicating the data was not accepted.
R-18	The oscilloscope host application shall allow the user to save captured sequences of oscilloscope samples to disk files in the comma-separated values (CSV) text format.

While this set of requirements may yet be incomplete, it provides a good starting point for the development of a comprehensive set of system tests.

Once the system requirements have been delineated, it is necessary to determine the approach that will be used to determine whether the system properly implements each requirement. There are four fundamental methods for evaluating whether a system complies with a particular requirement:

- **Inspection:** Some requirements can be verified by simply examining the system. For example, a requirement that the device be yellow in color can be verified by inspection.
- **Demonstration:** System features that can be observed by operating the system are considered demonstrations of performance in terms of the relevant requirements.
- **Analysis:** Requirement verification through analysis relies on a logical deduction process that, given a particular set of facts that have been proven through inspection, demonstration, or testing, a given requirement has been satisfied.
- **Test:** Requirement verification through testing involves the operation of the system through a prescribed set of steps under controlled conditions. During the execution of these procedures, sufficient data must be collected to demonstrate the degree to which the system complies with the relevant requirements.

Our focus in this chapter is on the use of testing to verify embedded system performance in terms of system requirements. Testing against system requirements must take place under nominal conditions as well as off-nominal conditions, as discussed in the next section.

Testing under nominal and off-nominal conditions

Testing under nominal conditions serves to validate whether the system performs as expected when presented with correct inputs while operating under ordinary conditions. System tests should cover the entire allowed range for each input to determine whether the smallest and largest values for each input parameter result in correct operation. In addition to providing minimum and maximum values for each input, a selection of values across the range between the extremes should be tested as well, including any values that may result in special treatment, such as zero.

In deciding how many values within an input's range to evaluate during testing, it is important to weigh the value of testing the parameter more thoroughly against the costs of additional test runs, which will include the time to develop the test as well as its execution time (whether performed manually or automatically) and the time required to assess the results to determine whether the system is behaving correctly.

During nominal-condition testing, it is also important to consider combinations of inputs that may interact to introduce potentially unintended variations in system operation. All such parameter interactions should be considered and prioritized to determine whether testing each combination is warranted.

Off-nominal testing involves operating the system outside of its intended conditions to understand how the system reacts. For example, in the case of our oscilloscope requirements, we would want to test the system's response to input voltages outside the intended range. With an input voltage slightly outside of the allowed range, we expect the system to operate normally, perhaps with an invalid measurement reading due to exceeding the ADC input range. Voltages further outside the specified range will stress and eventually exceed the limited electrical protection capability of our circuit design.

To assess the safety of the system, it may be necessary to provide input voltages far outside the expected range because the possibility exists that users of the device may intentionally or accidentally connect the system to excessive voltages. Such voltages might include the 110 V and 220 V **alternating current (AC)** voltages available at electrical outlets in most homes and office buildings. Although providing these voltages as input to the device may cause serious damage to it, the system's response should not endanger the user by transferring the high voltage to a user-accessible area (assuming the circuit board is installed in a protective case) or by catching on fire or emitting hazardous fumes.

Testing electronic devices for approval by electrical safety certification authorities is complex and is outside the scope of our current discussion. The goal for the test process discussed in this chapter is to ensure the system performs its intended functions under nominal conditions and, when operated under reasonable off-nominal conditions, it responds in an appropriate manner. If our testing is adequate, we should avoid any need for drastic responses to user complaints, such as product recalls or the rushed delivery of software patches to the field.

The next section will discuss the distinction between unit testing and functional testing.

Unit testing versus functional testing

In the *Test-driven development* section of Chapter 9, *The Firmware Development Process*, we introduced an approach to firmware development that involves developing and running tests on source code as the code is being developed. This testing necessarily begins at the lowest-level units of code, which in C/C++ consists of functions.

Testing at this level is called unit testing. **Unit testing** attempts to verify that each function and each line of code within it performs as intended without undesired effects given all possible combinations of inputs. A development process that relies on TDD typically results in a collection of unit tests that is comparable to the application code in terms of the number of lines of code.

Through the use of testing frameworks such as Ceedling, also discussed in *Chapter 9, The Firmware Development Process*, the unit testing process can be automated to a high degree, which allows frequent test execution. By re-running tests often, developers can detect when a change to code results in incorrect behavior and fix the problem immediately. This prevents many bugs from entering the application code base that would have gone unnoticed in a more traditional development process.

Unit testing, however, can only get us so far. As we combine low-level code components into subsystems, it becomes less feasible to perform automated testing on these higher-level features. In the case of our digital oscilloscope, the behavior of the system relies on sophisticated hardware and FPGA firmware, which is often difficult to simulate in an automated software-based test.

Testing at a level that enables evaluation of an application's primary features is referred to as functional testing. In **functional testing**, users (or simulated users) interact with the system to exercise its primary features. Functional testing is the type of testing we use to evaluate system performance relative to its requirements.

Unit testing is based on a **white box testing** approach, where all aspects of the code are accessible to the tester. Functional testing typically ignores system internal implementation details and just looks at its behavior in response to test inputs. This approach is referred to as **black box testing**.

The following table summarizes the differences between unit testing and functional testing:

Attribute	Unit testing	Functional testing
Test purpose	Testing individual code components	Testing system features
Test focus	Individual function	Entire system
Test approach	White box testing	Black box testing
Level of automation	Highly automated	Limited or no automation
Types of problems detected	Code logic errors, off-by-one loop count errors, incorrect branch conditions, and the like	Functionality problems
Coverage metric	Number of lines of code tested	Number of requirements tested
Number of tests	Many	Limited
Cost and schedule allocation	Low	High

It is important to understand that the distinction between unit testing and functional testing is not a decision to perform one type of testing or the other. Rather, for a successful development effort, it is important to combine both forms of testing at a level dictated by the needs of the system design and development process.

Unit testing should be employed throughout the code development phase of the product development cycle. Functional testing only becomes possible when system development has reached a point where representative portions of the final intended system functionality are available for testing.

The next section introduces the concepts of negative testing and penetration testing.

Negative testing and penetration testing

The earlier discussion addressed the need to conduct system testing under off-nominal conditions. Tests that intentionally drive invalid inputs into the system are considered **negative testing**. The goal of negative tests is to assess whether the system correctly rejects invalid inputs and responds with appropriate error messages or other feedback to the user.

Negative testing does not only encompass erroneous inputs produced by users of the system. It can also include deliberate attempts to create problems with the system's operation and can represent attempts by unauthorized users to gain access to a system and exploit it for their own purposes.

Because our system is network-enabled, it is certainly open to attacks of this type over a network shared by multiple users and perhaps connected to a wider network such as the internet.

Negative testing that represents unauthorized users attempting to gain access to a system and then either subvert its proper operation or extract sensitive information is referred to as **penetration testing**. Any system that provides a digital interface, whether that connection is through a medium such as the **Universal Serial Bus (USB)** or directly to the internet, perhaps via Wi-Fi, should undergo comprehensive penetration testing to evaluate system security in the presence of realistic hacker threats.

Some systems operate in the context of hardware that is so complex that the only way for the system to function outside its intended operating environment is to surround it with a simulated environment. This is the topic of the next section.

Testing in a simulated environment

Consider the development process for an attitude control system used by a communications satellite that will orbit the Earth. This is a complex real-time embedded system that must perform a variety of sophisticated operations to keep the satellite's communication antennas oriented properly toward the Earth, maintain alignment of the solar panels toward the sun, and minimize the consumption of thruster propellant over time.

Obviously, this system cannot undergo its initial testing in the intended operating environment, which is in orbit around the Earth. To perform functional testing in a ground-based environment, it is necessary to generate simulated inputs to the attitude control system that are representative of the environment in which it will operate. Outputs from the controller, for example commanding a thruster to fire for a period of time, must drive a mathematical model of the system to represent its response to the controller output.

The model used to represent the dynamic system in this type of simulation is typically represented as a system of differential equations, the discussion of which is well outside our scope in this chapter. Even so, you should be aware that simulation-based system testing is used widely in the development of control systems for complex systems such as automobiles, aircraft, and spacecraft.

The next section will discuss some considerations that may be helpful in the development of repeatable test procedures.

Achieving repeatable test results

One of the key attributes of a valid system test is its repeatability. If it is not possible to repeat a test and achieve the same results, the test has no enduring value. If you can't repeat a test and get the same result, you will not be able to tell whether an intended fix implemented in the system has actually solved a problem.

To ensure test repeatability, it is important to understand all of the factors that determine the results of a test, and then control each of those factors during the test.

For example, if a particular test relies on the system processor being loaded to a certain degree with particular types of processing activities, it must be possible to recreate a similar processing load each time the test is run.

Of course, it will not always be possible to control each relevant factor when conducting a test. Depending on the type of system you are developing, system-level testing may take place in the presence of variable factors, such as the weather. Under these circumstances, it may be necessary to employ statistical techniques to gain an understanding of the system's response to factors outside your control.

It may be the case that a particular bug only manifests itself under a very specific set of circumstances, which you cannot entirely control. If you run into this kind of a problem, you have my sympathy. Perhaps the only way to get the problem to repeat itself is to control as much as you can and repeatedly execute the test in an attempt to recreate the problem.

If you end up performing repetitive tests in an attempt to flush a bug out into the light, be certain you are collecting all of the data on each run that might help you trace the problem to its source. Few things are as disappointing in testing as completing a difficult test successfully and then realizing you forgot to start your data collection tools at the beginning of the test run.

The next section will discuss the importance of developing a comprehensive test plan.

Developing a test plan

When preparing to conduct a series of tests to validate a system's performance against its requirements, it is important to spend a bit of time planning out the tests that will be performed. The planning process should be documented in a test plan that describes the tests to be performed, how data will be collected and analyzed, and how the analysis will result in a determination of the system's compliance with its requirements.

A test plan does not need to be a large, formal document. The focus should be on clearly defining the testing to be performed and the degree to which performance against system requirements will be validated. The plan should also identify any safety hazards that may arise during testing and clearly explain how the risk of personal injury and damage to valuable equipment will be mitigated to an acceptable level.

Stakeholders in the ultimate success of the system should be given an opportunity to review and comment on the plan before testing commences. The goal of the test planning process is to achieve consensus among all concerned that the series of tests will provide definitive value in validating system performance relative to the requirements while holding the cost and duration of testing within acceptable limits.

The following outline lists the sections contained in a typical test plan:

Section	Contents
Introduction	Briefly describe the system being tested and the purpose of the testing.
Testing scope	Identify which portions of the system requirements are to be covered by the testing and which, if any, are excluded.
System requirements	List the system requirements that will be evaluated during the testing
Test environment	Identify the location where testing will take place and describe the features of the location relevant to the testing.
Test schedule	Identify the planned start and end dates of testing. If testing will take place in multiple phases, identify the start and end of each phase.
Test resources	Identify specific equipment and personnel required for testing.
Risks	Identify any potential risks of damage to equipment or harm to personnel. Estimate the likelihood of occurrence for each risk and the consequences if the risk is realized. Identify plans to mitigate these risks.
Test team members	Identify the team members who will participate in the testing and specify the leader for test activities.
Test descriptions	Provide an overview of each test to be performed. Detailed test procedures may be provided in an appendix or as a separate document.
Data collection plan	Describe the types of data to be collected and the tools that will be used to collect the data. Describe the procedure for operating the data collection tools during testing and how data will be gathered at the completion of testing.
Deliverables	Describe the content of the test report to be produced after testing is complete.
Signatures	Provide signature blocks for appropriate technical and managerial leaders.

To reiterate, the test plan does not need to be a large, complicated document. For a small system, the information listed in the preceding table might be laid out in a two-page document or even in an email. The purpose of conducting the test planning exercise is to think through each of the topics in the plan to ensure nothing important is left out and to confirm that all participants and stakeholders have a good understanding of the testing process.

In the next section, we will discuss some specific recommendations for conducting tests and recording test results.

Conducting tests and recording results

It is best to plan the details of complex tests ahead of time and to lay out written procedures for conducting tests that involve multiple participants. The following sections present a few recommendations for achieving success when testing complex systems.

Identify the data to be collected

Once the structure of a test has been articulated, the tester must determine what information must be gathered during and after the test to answer any questions that may arise during analysis of the test results. This information will naturally include data from the system itself that represents its behavior, including information displayed to the user or recorded as part of normal system operation.

With our focus on embedded systems, it is likely that additional information will need to be collected to evaluate the results of some test scenarios. This data may be gathered using a wide variety of techniques. Some examples of data collection tools and strategies for testing embedded systems are listed here:

- **Network packet capture:** Tools such as Wireshark (<https://www.wireshark.org/>) are capable of capturing every packet traversing a network. This data can be used to determine precisely what data a system sent and received over a network and the time at which each packet was captured.
- **Analog voltage measurement:** Analog system inputs and outputs provide raw material for understanding the system's responses to those signals. Analog signals may need to be recorded continuously during testing, such as with audio inputs or outputs. Alternatively, it might be necessary to capture analog signal behavior in response to specific triggering events, perhaps using an oscilloscope.

- **Digital signal capture:** One or more digital signals within the system under test or accessible at external connectors can provide crucial information about system behavior. **Logic analyzers** are tools that monitor a potentially large number of digital signals simultaneously and can capture those signals at high sample rates. Logic analyzers enable the monitoring of system data buses and digital communication protocols.
- **Video recording:** Sometimes the most straightforward way to capture system behavior during testing is to use one or more video cameras to form a visual record. When using video recording during testing, it is vital to provide some means of synchronizing the time of observations in the video with data collected from other sources. This may be as simple as carefully synchronizing the video camera's clock and ensuring a timestamp appears in the video. Alternatively, you might place an accurate clock display in the video scene along with the system under test.

To ensure comprehensive data collection, and to avoid the need to repeat tests due to missing data, it is vital to ensure that all data collection mechanisms planned for each test are recording at the beginning of each test.

Next, we will discuss the need to ensure the system is configured properly at the start of each test.

Configuring the system under test

All aspects of the system configuration that may affect the results of a test must be defined as part of the test procedure and confirmed to be set as intended before starting the test.

It is frustrating for everyone involved if some form of anomalous behavior is observed in the results of a test and the tester is unable to answer questions about configuration settings in the system that may have produced the observed behavior.

Test procedures should contain a list of steps to perform system configuration at the start of each test. This list should include all of the settings that deviate from default values, as well as any values that may have been changed during the conduct of an earlier test.

With the system properly configured, and all data collection systems activated, it is time to begin executing the test procedures.

Executing the test procedures

Professional testers conducting high-cost, high-stakes testing of complex systems such as aircraft and spacecraft typically document the detailed steps in a test procedure on one or more cards containing detailed instructions for each test participant.

Depending on the type and complexity of your tests, it may be worthwhile to prepare a set of written test steps ahead of time. It is helpful to have the steps laid out in an easy-to-follow format for tests that involve multiple participants performing a coordinated sequence of actions.

The use of written test procedures avoids some of the delays and confusion that might occur during testing if, for example, some form of mental calculation would otherwise be required to determine an input value, or if you need to locate a file in a rarely accessed directory.

Each step on the card should be numbered, and the test leader should indicate steps by number as the test progresses to keep all participants synchronized. Before initiating a test, the test leader must ensure any safety criteria related to the test have been satisfied. The leader should state when data collection systems should be started, when the test execution starts, when the test ends, and when to cease data collection. After each test is completed, data collected during the test must be stored and labeled to ensure it can be retrieved for analysis.

Although the detailed behavior of the system during each test will not be known until analysis of the data has been completed, it is nevertheless possible to make tentative statements regarding system performance during the test based on real-time observations by the testers. This is called a quick-look assessment.

Quick-look assessment of test results

A quick-look assessment provides impressions of system performance immediately following a test to stakeholders in the system development process. A quick-look assessment may involve nothing more than asking each test participant to state what he or she observed during the testing and to indicate whether the system appeared to function in a manner consistent with relevant system requirements.

During the quick-look assessment, testers should describe any anomalies they observed during the testing. It is not helpful to extrapolate in suggesting the underlying cause of observed behavior if there is no data immediately available to support that conclusion.

All participants in the quick-look assessment must understand that the observations presented are tentative and are subject to change during later analysis of data collected during the test.

One outcome of a quick-look assessment may be a determination that the test should be repeated. This possibility is the topic of the next section.

Repeating tests when necessary

One result of a quick-look assessment should be a determination as to whether the test was conducted correctly and the necessary data was collected. If significant errors in the conduct of the test are found, or some data did not get captured, it may make sense to repeat the test.

The feasibility of repeating a test soon after the completion of a faulty attempt to perform the same test depends on the availability of test resources, test personnel, and the necessary time to prepare for another test attempt and then conduct the test.

It may be advantageous to follow a failed test attempt with another test run if everything remains in place and the people needed to conduct the test are available. Of course, the ease of conducting additional tests should not drive a decision to repeat a test if it is not truly necessary.

If repeating a test is not a straightforward exercise, perhaps because preparing for and executing the test again would extend the test schedule to an intolerable degree, the test team may need to make a judgment call as to whether the initial test data is suitable for assessing system performance, even in the presence of test execution errors or gaps in the data.

This section and the preceding sections have focused on system-level tests and the complexities that may arise with planning and conducting testing of the entire system. The next section will look at a different aspect of testing: The need to perform regression testing on previously tested code after changes have been made to the code.

Regression testing existing code

As anyone who has learned the rudiments of coding computer software knows, code is extremely brittle. If one were to select a single non-comment character at random in a file of computer code and change it to a different character, it is very unlikely that the program would continue to perform correctly, if it could be made to run at all.

The act of thoroughly testing a piece of computer software running on its intended hardware imparts substantial value to the code. Not only does the software work correctly, you *know* it works correctly, at least under the tested conditions.

After the initial test program has completed and the product has been released, there will very likely be ongoing reasons to modify the code, either to fix problems that remained after the completion of testing or to add new features.

To maintain the value of the previously developed and tested code, it is vital to ensure any changes introduced during maintenance and feature enhancement do not introduce errors that cause previously functional code to misbehave. One aspect of code brittleness is that a change that seems *far away* from influencing a particular section of code can cause errors that manifest in unexpected ways in that code.

The purpose of regression testing is to regularly test existing, working code to ensure it continues to perform correctly in the presence of changes in other portions of the code base. The primary mechanism for performing regression testing is the reuse of unit tests created during code development. The entire collection of unit tests must be maintained, version controlled, and updated as changes and additions are made to the system code base.

Each change to system code should take place in the presence of tests that verify the correct behavior of those changes. In addition to newly created tests related to changes being introduced to the code, it is necessary to frequently re-run all of the tests for code that interacts in any way with the modified or new code.

For a system of substantial code size, with a large collection of tests, execution of the entire test suite may take a significant amount of time. The planning and coordination of frequent runs of the full test suite should be integrated into the software development process in a manner that minimizes delays and interference between the testing and ongoing development work.

The need to perform frequent runs of the full test suite might be satisfied by scheduling these test runs overnight or on separate computer systems from those used by the developers.

When designing a suite of tests, it is vital to ensure everything that needs to be tested gets tested. This is the subject of the next section.

Ensuring comprehensive test coverage

To ensure that nothing of critical importance is overlooked when designing a suite of tests for your system, it is necessary to approach the design of the tests in a methodical manner. The use of appropriate metrics will tell you how well your tests cover the aspects of the system that require testing.

One important metric is the degree to which the test process assesses system performance against each requirement. Another metric, focused on the software and firmware portions of a system design, evaluates the degree to which the test suite covers the various flows of execution through the code. The next section addresses the coverage of system requirements during testing.

Requirements traceability matrix

A **requirements traceability matrix (RTM)** is a table that documents the test cases that address each system requirement. By developing an RTM during the test design process, you can determine which of the requirements have been validated by test cases and which have not.

Let's examine a small subset of the tests we might develop to assess a system prototype against the requirements for our oscilloscope. These tests are listed in the following table:

Test ID	Description
T-1	Use a signal generator to produce a constant output voltage of +10 V. Save 1.0 seconds of samples to a disk file. Verify that the oscilloscope measures +10 V within 0.05 V, 95% of the time.
T-2	Use a signal generator to produce a constant output voltage of -10 V. Save 1.0 seconds of samples to a disk file. Verify that the oscilloscope measures -10 V within 0.05 V, 95% of the time.
T-3	Use a signal generator to produce a 10 KHz sine wave with an amplitude of 10 V. Save 1.0 seconds of samples to a disk file. Verify that the oscilloscope measures within 0.05 V of an idealized 10 KHz sine wave, 95% of the time.
T-4	Use a signal generator to produce a 1 MHz sine wave with an amplitude of 10 V. Save 1.0 seconds of samples to a disk file. Verify that the oscilloscope measures within 0.05 V of an idealized 1 MHz sine wave, 95% of the time.
T-5	Set a rising edge trigger at +5 V. Use a signal generator to produce a 10 KHz sine wave with an amplitude of 10 V. Save 1.0 seconds of samples to a disk file. Verify that the oscilloscope triggers at the first sample that is greater than or equal to the +5 V point on the rising slope of the sine wave.
T-6	Set a falling edge trigger at +5 V. Use a signal generator to produce a 10 KHz sine wave with an amplitude of 10 V. Save 1.0 seconds of samples to a disk file. Verify that the oscilloscope triggers at the first sample that is less than or equal to the +5 V point on the falling slope of the sine wave.
T-7	Repeat test T-5, this time configuring the oscilloscope to store 0.65 seconds of data prior to the trigger and 0.65 seconds of data following the trigger. Save 1.3 seconds of samples to a disk file. Verify that the trigger is at the correct location as defined in test T-5 and the correct number of samples are recorded before and after the trigger event.
T-8	Use a signal generator to produce a ± 5 V square wave at a 100 Hz frequency. Configure a rising edge trigger at 0 V with a minimum pulse width of 10.1 ms. Verify that the oscilloscope does not trigger on the square wave signal.
T-9	Use a signal generator to produce a ± 5 V square wave at a 100 Hz frequency. Configure a rising edge trigger at 0 V with a minimum pulse width of 9.9 ms. Verify that the oscilloscope triggers properly on the square wave signal.

A simple RTM consists of a list of the system requirements with two columns appended. The *Verification method* column documents which of the verification methods (inspection, demonstration, analysis, or test) is appropriate for evaluating the system's implementation of the requirement. For each requirement with the *Test* verification method, the *Test cases* column contains a list of tests that will assess that requirement.

We will use the information in the system requirements (as listed in the table in the earlier *Requirements-driven testing* section) and in the set of test cases (in the preceding table) to construct an RTM that correlates test cases with the system requirements that each case assesses. The following table contains an example RTM based on this information:

Requirement ID	Description	Verification method	Test cases
R-1	The oscilloscope shall have a single input channel consisting of a BNC connector for use with standard oscilloscope probes.	Inspection	
R-2	The oscilloscope shall sample its input signal at a 100 MHz rate.	Test	T-3 T-4
R-3	The oscilloscope shall support input signals over a range of ± 10 V when using a 1X oscilloscope probe.	Test	T-1 T-2
R-4	The oscilloscope shall support input signals over a range of ± 70 V when using a 10X oscilloscope probe.	Test	
R-5	Oscilloscope samples shall contain 14 bits of resolution.	Test	T-3
R-6	The oscilloscope shall collect and store up to 1.3 seconds of continuous measurements.	Test	T-6
R-7	The oscilloscope shall transfer collected data to a host application over a TCP/IP-based network.	Test	T-3 T-4 T-5 T-6
R-8	The oscilloscope shall support rising and falling edge-based triggering.	Test	T-5 T-6
R-9	The oscilloscope shall support pulse width-based triggering.	Test	T-8 T-9

R-10	The oscilloscope shall store a user-selected number of pre-trigger samples.	Test	T-7
R-11	The oscilloscope shall store a user-selected number of post-trigger samples.	Test	T-7
R-12	The oscilloscope shall receive and respond to user command information received over the network.	Test	T-1 T-5 T-8
R-13	The oscilloscope shall tolerate normal handling of the signal input connection, including static electric discharge.	Test	T-1 T-5 T-8
R-14	The oscilloscope shall operate over a temperature range of 0 to 70°C.	Test	
R-15	The oscilloscope shall operate over a relative humidity range of 8 to 80%.	Test	
R-16	The oscilloscope shall maintain a collection of status information and provide this data to the host application upon receiving a request.	Test	T-5 T-5 T-8
R-17	The oscilloscope shall ignore invalid or out-of-range command data and set a status indicating the data was not accepted.	Test	
R-18	The oscilloscope host application shall allow the user to save captured sequences of oscilloscope samples to disk files in the CSV text format.	Test	T-1 T-3 T-5

When several test cases are equally applicable to a particular requirement, it is not necessary to conduct analysis of each test case against all potentially relevant requirements. It is sufficient to associate the minimum number of test cases to each requirement to evaluate if system performance satisfies the requirement.

The preceding table indicates that the set of tests listed in the earlier table does not address all of the requirements. Requirements R-4, R-14, R-15, and R-17 remain untested. This highlights a primary benefit of constructing an RTM: it shows you which requirements will be tested and which will not be tested given a defined set of tests. Now we know the areas in which additional tests must be developed if we are to address all of the system requirements.

The next section will focus on approaches for thoroughly testing code for the system firmware and software.

Tracking code coverage

Between the unit tests created during the development of system code and the system tests that evaluate performance in terms of requirements, we should already have achieved good coverage of the code. However, we do not yet know the degree to which each possible path through the code has been exercised by the tests.

Code coverage describes the degree to which a program's source code is exercised by running one or more tests on it. There are several categories of code coverage metrics that assess the thoroughness of testing. The following list describes the major types of code coverage testing in order from least to most thorough:

- **Statement coverage:** Statement coverage is the percentage of source code statements that are executed by the test suite. Many unit test frameworks and other types of software testing tools provide the ability to produce a report of statement coverage following completion of a test set. The use of statement coverage as a code coverage metric has the drawback that, even if every statement in a program is executed during testing, not all paths through the code have necessarily been tested.
- **Branch coverage:** Branch coverage testing attempts to ensure that every branch in the code is taken. To demonstrate the difference between statement coverage and branch coverage, consider the following section of C code:

```
if (AdcReading > MaxReading)
{
    HandleReadingOutOfRange (AdcReading) ;
}
```

By creating a test case that sets `AdcReading` greater than `MaxReading`, a single test case will provide 100% statement coverage of this code but only 50% branch coverage. An additional test case with `AdcReading` less than or equal to `MaxReading` is needed to cover the alternate branch path.

- **Condition coverage:** Condition coverage testing ensure that not only are all branches covered, but each component of branch conditions is evaluated as well. This evaluation applies to conditional expressions containing multiple elements that can each result in a true or false Boolean value.

Consider the following C code:

```
if ((AdcReading > MaxReading) || (AdcReading < MinReading))  
{  
    HandleReadingOutOfRange(AdcReading);  
}
```

To achieve full condition coverage for this code, each Boolean subexpression in the conditional expression must be tested to return both true and false results. To achieve 100% condition coverage for this code segment, three test cases are required: The first sets `AdcReading` greater than `MaxReading`, the second sets `AdcReading` less than `MinReading`, and the third sets `AdcReading` between `MinReading` and `MaxReading`.

The types of code coverage analysis listed here are normally performed automatically during unit testing. It is generally not feasible to verify test case coverage using manual analysis methods alone. This means that tool support for the different forms of code coverage assessments described in this section should be under consideration when selecting a set of test tools for use in unit testing.

You may also need to perform some degree of code coverage testing as part of system-level testing. Evaluating code coverage during system tests may be far more difficult due to the need for code to execute in real time in the embedded system. You may be able to execute system tests with a debugger attached to the system under test. This allows you to monitor code execution in real time. While performing testing with a debugger attached to the system can provide detailed information about the flow of execution, you must ensure that the presence of the debugger does not introduce any artificiality into the behavior of the system under test.

The next section will discuss an approach you can use to determine how much testing of the system is enough.

Establishing criteria to define how much testing is sufficient

After developing an RTM and correlating tests to be conducted with the requirements each test would verify, the next step is to determine which of the tests must be executed. It may turn out that particular tests are very expensive, time-consuming, or have resource requirements that cannot be readily satisfied. For such tests, it is necessary to weigh the value of conducting the test against these costs.

For example, in the case of requirement R-15 in the preceding table, testing the oscilloscope under varying conditions of controlled humidity may be infeasible because conducting such testing would require paying a testing laboratory to perform the evaluation. If the cost of such testing is not in the project budget, it might be acceptable to change the verification method for R-15 from Test to Analysis.

The analysis required to support this change might look at the humidity specifications of all the components in the device and combine that information with other factors, such as the seal quality of the device case, to provide evidence that the assembled product will meet the humidity requirement. This form of analysis may or may not be acceptable for a given product depending on whether formal certification of compliance with the requirements is needed. These certification processes sometimes mandate testing to demonstrate the product is in full compliance.

After selecting the set of tests to be performed and determining them to be executable within the cost, schedule, and resource constraints of the project, the time has arrived to conduct the testing, gather the resulting data, analyze the results, and assess the performance of the system against its requirements.

If all the tests pass, great! It is common, however, to identify problems with the system in the course of testing. As these issues are identified and documented, a decision process must take place to determine whether to proceed with testing in the presence of the now-known problem or to stop testing and try to fix the problem before continuing with testing.

Whether the test program pauses in response to a serious test failure or continues uninterrupted, at some point it will be necessary to identify the source of the problem and address it. Once work gets underway to address an identified system deficiency, the first steps are to understand the full scope of the problem, trace it to its source, and develop a fix or at least a way to work around the problem. These topics are the subject of the next section.

Errors in the design or implementation of hardware or software in a system are commonly called **bugs**. The process of finding and removing these errors is called **debugging**. Bugs can range in severity from the nearly negligible, such as a spelling error in help text displayed to a user, to the catastrophic, such as a flaw in an aircraft autopilot system that renders the aircraft uncontrollable under specific flying conditions, causing it to crash into the ground.

Bugs can be detected and fixed at any point in the design, development, and testing processes for a complex system. The following sections discuss some approaches that are helpful for detecting and eliminating bugs in hardware and software.

Dealing with syntax and compilation errors and warnings

While seasoned software developers are generally able to quickly identify the source of the problem when a programming language compiler reports syntax or other forms of errors, junior developers may struggle to understand the cause of the problem and then implement an appropriate solution when such issues arise. The developer must take care to ensure that the solution is correct in terms of the overall system implementation and is not merely a quick solution that stops the compiler from complaining.

For example, if the compiler reports that a variable is used that has never been defined, the developer must first determine whether the referenced variable is actually needed or if it is instead a misspelling intended to reference an existing variable name.

If a new variable must be defined, the data type must be selected among the choices of integer or floating point, number of bits, and signed or unsigned for integer types. Finally, the variable must be defined at the proper scope. Should it be local to a statement block contained between curly braces? Should it be local to the enclosing function? Should it be a static global variable scoped to the current source file? Or should it be declared global and be accessible from code anywhere in the program?

Some types of syntax errors can be challenging to correct. As another example, if a closing curly brace is missing from a section of code that contains multiple levels of blocks within blocks, determining the correct location to put the missing brace may require extensive examination of the surrounding code. Inserting the brace at the wrong location might cause the compiler error to go away, but it would also drastically change the behavior of the code.

To summarize, developers should pay close attention to error messages emitted by a compiler and carefully implement solutions to resolve the problems.

Developers should also strive to eliminate warning messages produced by the compiler. If compiling your application produces a stream of warnings, it is easy to miss important warnings among the messages you might think are frivolous. If you take the time to understand the reason a particular message is produced, you may come to realize fixing the issue causing the message is worth the effort.

In the case of warning messages that you determine to be truly irrelevant, there is usually a way to suppress individual message types using compiler configuration settings or command-line options when building code. When finished, your code should generate zero errors and zero warnings during compilation.

The use of static code analysis can effectively extend beyond the code parsing capabilities of the language compiler to perform a much more in-depth analysis of your code, identifying many types of potential problems that the compiler ignores. This is the subject of the next section.

Working with static code analysis and unit testing

In the *Statically analyzing source code* section of *Chapter 9, The Firmware Development Process*, we examined in some detail the purpose and use of static source code analysis tools. When static code analysis is fully integrated into a software development process, the code analysis tools run frequently, perhaps as often as every time the compiler processes a source file.

By incorporating the automated execution of static code analysis into the software build process, you gain the benefits of static analysis without the need to add more steps to your edit-compile-test cycle.

By performing code analysis frequently, you can identify issues while the code you are working with is fresh in your mind and the extent of changes since the preceding analysis is small. If you instead implement a large number of changes across the application code base before running static analysis, you may generate a large number of messages that require extensive code review to understand and resolve.

A drawback of frequent executions of the static code analysis tool is that the analysis process takes some time. Many static code analysis tools provide a mechanism for generating and storing analysis results at the granularity of individual source files. With these tools, the speed of the analysis is greatly increased because the process requires a full analysis only of files that have changed since the last set of analysis results was generated. The analysis results for files that have not changed are directly available in the stored data.

As with the compiler warning messages discussed in the previous section, you should strive to produce code that generates zero static analysis messages. When necessary, use the configuration options for the static analysis tool to suppress messages that are truly irrelevant to your code.

Code that builds with zero compiler warnings and zero static analysis warnings can still contain a lot of bugs. The initial steps of the debugging process are discussed in the next section.

Define the problem clearly and attempt to duplicate it

When the results of a test indicate the system is behaving incorrectly even though there are no compiler or static analysis warnings and, as far as you know, the code implements the intended algorithm correctly, it is necessary to begin the debugging process.

Start by defining the problem in terms of differences between the expected behavior and the actual, observed behavior. Identify the *exact* indicators that provide clues that something is not right. Make note of any aspects of the test conditions that may have influenced the system to behave in the observed manner.

If possible, attempt to duplicate the erroneous behavior by re-running the test. If you can reliably repeat the erroneous behavior, you are already a long way down the road to resolving the problem.

If attempts to duplicate the problem are unsuccessful or intermittent, you face a more difficult situation. If the problem only occurs on some test runs but not on others, try to identify factors that may influence the system's behavior.

For example, it may be that the time between system power-on and the start of the test contributes to the problem in some way. Think about the things that change in the system as time passes: Timers count up to larger values and ICs become warmer. If these or similar concerns could possibly contribute to an observed intermittent problem, factor them into the test execution process and determine whether they are relevant.

If a problem only occurs once during a test run and you are unable to duplicate it, you may still be able to track it to the source using the data collected during the test run. This is another reason to double-check that all of your data collection tools are running at the start of each test: you may only get one chance to gather the data you need to fix a serious system problem.

If you are able to reliably duplicate the problem, you can begin an orderly debugging process to track the problem to its source. The first step is to verify that the inputs to the system are correctly set for the intended test conditions. This is the topic of the next section.

Determine whether the input is correct

Before you dig into the code and start trying to identify problems there, first verify that the test is set up correctly and the test procedure is providing the proper input to the system for the intended test conditions.

If the system is operating with an inappropriate configuration setting during a test, perhaps because the configuration change was left over from an earlier test, this can easily produce test results that appear erroneous. Examine all relevant system configuration settings prior to each test run to ensure configuration item that could possibly have an inappropriate value is properly set.

If you learn that the system was not configured correctly for a test, add steps to the test procedure to confirm that the relevant configuration settings are correct at the beginning of the test.

If the test procedure involves input from human operators, observe the operator interactions with the system during a test run to verify it is correct at each step. Sometimes users who are very familiar with a system will have a *muscle memory* approach for performing certain tasks that they may use unconsciously rather than strictly following the procedures laid out in a test card.

If input data intended for use in a test is organized in data files, try to find a way to verify that each element of the intended test data is actually being used correctly during interactions with the system under test. For example, if an automated test procedure attempts to log in to a system being tested with a username and password defined in a data file, look in the log files of the system under test to verify that the login occurred and that the username was correct.

If all of the test input data appears correct, and the steps of the test procedure are being followed properly, the next step is to gain a better understanding of the system's internal behavior in response to the input data. This is discussed in the next section.

Find ways to gain visibility into the system

At this point, we know the system is behaving erroneously during a test procedure and we believe the test execution and test input data are correct. We now need to get a better understanding of the system's internal operation related to the functionality under consideration.

Because we are working with an embedded system, this may be more challenging than traditional software debugging of a PC application. Many development tool suites for embedded processors include fully functional debuggers. A **debugger** is a software application that enables the control and monitoring of an application during execution, typically through a cable connected to the embedded system.

Debuggers generally provide the following features:

- **Source-level debugging:** A debugger that supports source-level debugging allows you to interact with the system in terms of your high-level source code statements and variable names. Debuggers also allow interaction with program code and data at the level of processor instructions, processor registers, and memory addresses.
- **Breakpoints:** A breakpoint is a location in a program's source code or a specific processor instruction at which code execution will stop when the flow of execution reaches that location. While execution is stopped, other debugger functions enable the display and modification of data in system memory and in processor registers. Debuggers for embedded systems may support a limited number of simultaneously active breakpoints.
- **Single-step code execution:** After program execution is stopped at a breakpoint, the debugger allows program execution to continue one source code statement at a time under user control. After each step, the displayed values of program variables and processor registers will be updated to reflect any changes resulting from the execution of the statement.
- **Variable and memory display:** While program execution is stopped, typically when code execution reaches a breakpoint, the debugger displays the values of user-selected variables and areas of memory. The user can modify the contents of variables or other locations in memory and resume program execution.
- **Watchpoints:** A watchpoint is similar to a breakpoint, except a watchpoint monitors access to a data memory location rather than the execution of a code statement. A watchpoint is attached to a variable or memory address and triggers when the location is accessed. Watchpoints typically allow specific conditions to be attached, such as triggering only on a read or a write to the specified location.

While using a sophisticated debugger makes it straightforward to interact with code executing in an embedded system, this level of capability is not always available to the developer. Sometimes the only interface to an embedded system being debugged is a serial port connected to a terminal program running on the developer's desktop computer.

In this situation, the most readily available mechanism available for observing the internal operation of the code may be to insert strategically placed `print` statements in the code that send output to the serial port. While this approach allows the developer to display arbitrarily selected information during code execution, it requires modification of the code in the system under test, and these modifications may have a significant impact on system behavior, depending on the execution time and resource usage associated with the `print` statements.

In the end, it may be up to the cleverness of the developer to extract the necessary information during code execution to understand the sequence of actions taking place in the system under test. If a debugger is not available, and if the use of `print` statements to a serial port is also not feasible, it may be possible to temporarily repurpose one or more **light-emitting diodes (LEDs)** on the device to provide useful debugging information such as indicating branch paths taken through the code.

The use of a binary search process for identifying the source of a problem can be an effective debugging method, as described in the next section.

Using a binary search debugging process

The binary search is an efficient, classic method for locating a value in a sorted array. A similar approach can be used to locate the source of a problem in a large code base for a software application.

If you begin the debugging process with no idea where the problem lies, the first activities you undertake should focus on narrowing the location of the problem. Try to identify an approximate midpoint of the *space* in which the problem exists and perform a test to determine which side of that midpoint contains the problem.

One obvious dividing line in the problem space for an embedded system is to determine whether the problem is caused by software or hardware. If you have multiple hardware instances of the system under test available, it may be worthwhile to run the same test on different hardware. This test may detect component failures or incorrect assembly of the system hardware tested initially, but it would not detect an inherent design flaw in the hardware that exists across all instances.

If the problem appears to be associated with software, the next level of the binary search is to determine which large-scale component of the software contains the flaw. Performing this test may involve the use of a debugger or the insertion of `print` statements to a serial port to examine code execution behavior.

This approach can be continued through multiple steps to further divide the code into smaller sections until the problem is isolated to a finite region of the source code. At this point, it is up to the developer to analyze the observed behavior and locate the problem area within the code.

After the problem has been identified and fixed, it is vital to go back and remove any temporary changes that were made to the code during the debugging process.

Another approach to debugging is to temporarily disable portions of system functionality to zero in on the source of problems. This is the topic of the next section.

temporarily removing portions of functionality

If the activities taking place simultaneously in the system make it too complex to determine what is occurring in the code related to an observed problem, it may be possible to temporarily disable the code performing unrelated activities to permit a focus on the problem area.

In C or C++, one simple way to disable a contiguous segment of code is to place a line with the text `#if 0` at the beginning of the code block and the line `#endif` at the end of the block. These two preprocessor statements will effectively remove all intervening lines of code from the compilation process regardless of the types of comments the lines may contain.

By eliminating processing activity unrelated to the code that appears to be associated with the problem, it may be possible to focus on the source of the issue more easily.

It is important to remember that one or more portions of the code have been commented out. After debugging is completed, you must remove the preprocessor lines that commented out the code segments and verify that the fix remains effective with the previously commented code running again.

If commenting out sections of code does not permit you to identify the source of the bug, the next level of debugging involves cutting the entire program down to the smallest program that manifests the bug. This is discussed in the next section.

Make the smallest program that demonstrates the problem

If your debugging efforts have been unsuccessful to this point, you may want to reduce your code to the smallest program that demonstrates the problem. This is desirable if you intend to submit the program to the vendor of your development suite or post it on a public message board requesting assistance.

To take this step, you should copy the code for your program to a location distinct from your main development directory structure. Then, cut away all parts of the program that do not affect the presence of the observed problem. Ideally, you will be able to reduce the relevant code to a small region, perhaps a page or two if the code was printed out.

If you intend to share the problem code with a vendor or place it on a public board requesting help, you should carefully scrub any sensitive information from the code and associated data files.

If you are able to reduce the size of a program that demonstrates the problem to a few dozen lines, you may be at a point where you can resolve the issue yourself. If not, you can share it with others and request their suggestions for resolving the problem.

This section has provided an overview of approaches for debugging problems observed during the testing of a complex embedded system.

The next section provides a collection of recommendations for the successful execution of projects involving the development of high-performance embedded systems.

Summary of best practices for high-performance embedded system development

This section contains a series of recommendations intended to help developers of high-performance embedded systems achieve success. While by no means exhaustive, this list should provide a good starting point for your next development effort.

Designing for test

One of the most constructive things you can do during the design process for a complex embedded system is to, at all stages, include features that enable easy testing of the system. By doing this, system designers enable the efficient evaluation of system performance and the rapid completion of system testing.

Designing for test may involve the addition of test points on printed circuits or the use of processor or FPGA chips that support enhanced debugging features. Regardless of the specific technologies involved in the design-for-test process, the goal must be to make the internal behavior of the system as visible as possible during testing.

One potential drawback of a comprehensive design-for-test approach is that publicly released versions of the system may contain design-for-test features that enable malicious actors to gain undesired insight into the internal operation of the system. These concerns can be allayed by taking advantage of advanced capabilities of modern processors and FPGAs, such as requiring the entry of a complex password to enable use of the system debugger port.

Leave room for growth

When establishing fixed resources in a system hardware design, consider possible future growth requirements. This may involve designing more processing capability into the system than required by the initial design and making both volatile and nonvolatile memory regions substantially larger than required. In FPGA designs, you may decide to use a better-resourced FPGA device than one that has just enough capability to implement the design.

By leaving plenty of room for future expansion, you can extend the lifespan of the product and potentially customize it in ways that make it attractive to customers beyond originally targeted users.

Of course, including more hardware capabilities than the minimum required for the design will result in higher cost as well as, possibly, the consumption of more area on the circuit board and increased power consumption. Considerations related to the possible benefits of adding room for future growth should be weighed carefully against the costs.

Design hardware with future capabilities in mind

Beyond including room for expansion of processing power and memory space, the hardware design may include features that are not mandatory in the original design but are perceived as useful in a future upgrade of the system.

For example, although the current hardware design of our digital oscilloscope does not include it, we could add hardware to implement an analog trigger function. The present design uses digital triggering, which compares the sampled data values from the ADC against the trigger voltage. This trigger method samples the input voltage every 10 nanoseconds and uses those samples to evaluate the trigger condition.

With this design, it is possible for pulses shorter than 10 nanoseconds to momentarily exceed the trigger level but fail to activate the trigger because the samples do not happen to capture the pulse.

By using high-speed analog hardware to perform the trigger function, rather than relying on discrete samples taken every 10 nanoseconds, it is possible for the oscilloscope to accurately trigger on these narrow pulses.

The analog trigger hardware adds a **digital-to-analog converter (DAC)** to set a reference voltage as an input to an analog comparator to use while monitoring the input signal. The comparator has a digital output signal that changes state when the analog input signal crosses the reference voltage. The latch captures the pulse, even if it is very narrow, and provides the trigger signal to the rest of the system. *Figure 10.1* shows how an analog trigger function could be added to the digital oscilloscope hardware design using a 14-bit DAC:

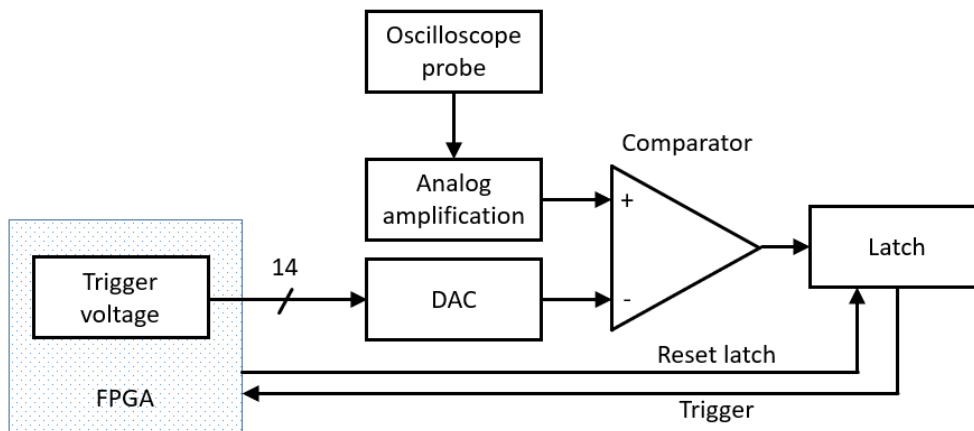


Figure 10.1 – Analog oscilloscope trigger circuit

For the current design of our oscilloscope project, we are using digital triggering, despite the limitations of this method.

If we had instead chosen to include the analog trigger hardware but not use it initially, a future firmware upgrade could then activate and use this hardware, thereby improving system performance.

In the next section, we will discuss the benefits of restricting coding activity to just the functionality currently being implemented.

Developing only the code you need right now

At each stage of code development, whether it is FPGA or embedded processor firmware, or application code for a program running on the host PC, developers should add only code for the functionality currently being implemented.

There is sometimes a temptation to add **hooks** or features to the code currently being worked on in anticipation of future enhancements, even though those features aren't needed for the capability being developed at the moment. When developers do this, the motivation is usually a desire to ease future work by getting things ready for it in the code.

This is often a bad idea because the addition of these hooks complicates the current development work and makes it harder to test the code, especially if the extensions are unrelated to the functionality being added right now.

Another reason to avoid inserting code in anticipation of future development is that things change. It may turn out that the feature you expected to add at a later date is never implemented for one reason or another. When that happens, the hooks you added are now useless vestigial artifacts complicating the code base and making it harder to achieve full test coverage.

The agile programming methodology of **Extreme Programming (XP)** boils the thought behind this section down to the adage *Do the simplest thing that could possibly work*. This means that, during code development, you should add only the absolute minimum code that fully implements the immediately required feature or capability and nothing more.

Maintain strict version control

The *Version control with Git* section of *Chapter 9, The Firmware Development Process*, briefly introduced the Git version control system. For any code development effort intended to have enduring value, it is vital to maintain strict version control from the beginning and at each subsequent stage.

With the appropriate use of version control, it is possible to precisely track the changes that have occurred in individual files and identify what changes were made in each version of a file.

The use of version control requires a certain level of discipline, especially if multiple team members are working on the same code base. There should be a central repository containing tested code that is easily accessible to all developers. This repository may be in a public location such as GitHub, or it may be a private repository located either at a commercial provider of Git hosting services or on a private server maintained within the development organization.

As developers add new code and make changes to existing code, versions that pass their local tests will be pushed to the shared repository. Each developer must frequently synchronize his or her local copy of the code with the shared repository. This brings changes other developers have been making into the local copy and enables the rapid detection of any incompatibilities with changes others have made.

A sufficient level of communication among developers must be maintained to minimize conflicts that may arise if multiple developers are working on the same areas of the system simultaneously. If two developers make changes to the same file at the same time, correctly merging those changes can become a significant headache.

Develop unit tests as code is in development

If developers employ a test-driven development process, the code pushed to the repository will have demonstrated correct functionality through testing. This represents a huge advantage over a more traditional development approach that shares code across the team that compiles and passes static analysis checking, but that has never been tested.

Completing static analysis without warnings should be considered a necessary but far from sufficient condition for pushing code to the shared repository. There are just too many ways that code that appears to correctly implement a given set of requirements may fail to do so in obvious or subtle ways. Unit testing can catch a high proportion of these problems, which greatly eases the integration of component-level modules into a set of system capabilities.

Employing a TDD approach to code development makes the thorough testing of code a natural part of the development process. Developers who have come to embrace the TDD approach tend to report a high level of feelings of accomplishment and satisfaction with their work. This positive feedback flows from the elimination of uncertainty about the quality of the code they develop at a very early stage in the development cycle.

Begin system-level testing as soon as functionality is present

In developing a new system design, much of the early coding work relates to basic, low-level capabilities associated with system hardware components such as I/O interfaces. As work proceeds, developers combine these low-level capabilities into subsystems that implement substantial elements of overall system capability. Once a sufficient collection of subsystems has been at least partially implemented, it becomes feasible to execute some subset of the full system capabilities.

When this stage is reached, it becomes possible to partially test the system's performance. This early testing may be informal, but it should still follow procedures laid out in test cases related to the available functionality.

Early testing enables the identification of system-level problems and performance limitations as soon as possible. By detecting these issues while code development is still in progress, it is easier to revisit already-developed code to implement a fix for an observed problem, assuming the debugging process leads to the identification of an unambiguous cause for the problem.

Early testing may make it clear that fundamental changes in the system design are required, perhaps involving the redesign of some portion of the hardware. This testing may also lead to a realization that one or more system requirements are unrealistic and must be revised. While discoveries like this may be painful, it is always better to learn about them earlier in the development process, rather than later.

Summary

This chapter provided a roadmap for planning and conducting thorough system testing in the context in which a device will operate and addressing problems identified during testing. To be sufficient, system testing must address the entire expected range of environmental conditions and input signals to evaluate operation under all conditions.

During testing, all inputs must be carefully recorded and any resulting behavioral anomalies must be noted in detail. Test repeatability is vital for fixing problems identified during testing. The chapter concluded with a summary of best practices for high-performance embedded system development.

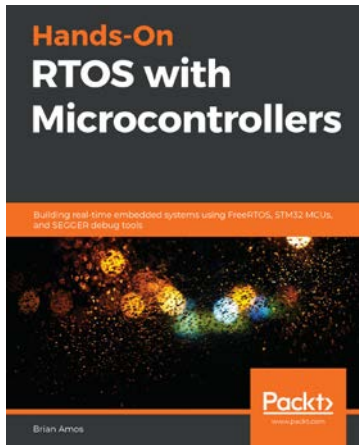
Having completed this chapter, you know the basics of efficiently and thoroughly testing a complex embedded system. You learned about running tests and recording test findings as well as techniques for efficiently tracking down bugs. You understand how to develop a set of tests that evaluate all important system aspects, and how to maintain focus on the key practices of successful embedded system development.

This brings us to the conclusion of the book. I hope you have enjoyed reading it as much as I have enjoyed writing it. We have covered a wide range of topics, beginning with the fundamentals of high-performance embedded systems. We moved on to the processes and tools involved in designing and constructing FPGA-based devices. We concluded with a series of chapters on all aspects of implementing, debugging, and testing real-time firmware. The information provided in this book should give you the background you need to begin designing and building high-performance embedded systems as you gain further knowledge on all of these topics from other books and from online information sources.

I wish you the best of luck in your embedded system development endeavors!

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:



Hands-On RTOS with Microcontrollers

Brian Amos

ISBN: 978-1-83882-673-4

- Understand when to use an RTOS for a project
- Explore RTOS concepts such as tasks, mutexes, semaphores, and queues
- Discover different microcontroller units (MCUs) and choose the best one for your project
- Evaluate and select the best IDE and middleware stack for your project
- Use professional-grade tools for analyzing and debugging your application
- Get FreeRTOS-based applications up and running on an STM32 board



Modern Computer Architecture and Organization

Jim Ledin

ISBN: 978-1-83898-439-7

- Get to grips with transistor technology and digital circuit principles
- Discover the functional elements of computer processors
- Understand pipelining and superscalar execution
- Work with floating-point data formats
- Understand the purpose and operation of the supervisor mode
- Implement a complete RISC-V processor in a low-cost FPGA
- Explore the techniques used in virtual machine implementation
- Write a quantum computing program and run it on a quantum computer

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

Index

Symbols

µc/OS-III
about 84
reference link 85

A

accurate temperature control 221
active sensor 31
ADC interfaces
 analog input 253
 high-speed digital interface 253
 SPI configuration port 253
Advanced eXtensible Interface (AXI) 283
analog multiplexer 33
Analog signals 48, 49
analog-to-digital converter (ADC) 32
 applying 32-36
analyzer output messages
 resolving 302
application-specific integrated
 circuit (ASIC) 8
Arduino
 URL 47
Arty A7 boards
 reference link 106

audio sensors 39

B

baseline Vivado project 165-176
billion samples per second (GSPS) 49
Bill of Materials (BOM) 231
binary semaphore 72
bitshift commands 273
black box testing 314
block diagrams 98, 99
block RAM 91
Block RAM (BRAM) 21
Board Support Package (BSP) 81, 254
branch coverage 327
bugs 329

C

cache memory 21
C/C++ 99, 100
Ceedling
 reference link 305
central processing units (CPUs) 6
chemical sensors 40
chip select (CS) 51

- circuit board
 - prototyping 210, 211
- circuit board assembly
 - checklist 231, 232
 - parts, placing 230, 231
 - prerequisites 230, 231
 - procedures 214
 - tools 214
- Clang-Tidy
 - reference link 298
- code coverage
 - about 327
 - tracking 327, 328
- code coverage, types
 - branch coverage 327
 - condition coverage 327
 - statement coverage 327
- code styling
 - about 291
 - automated code formatters 297
 - braces 292
 - comments 292
 - constants 296
 - implementation-defined
 - behavior, avoiding 294
 - indentation 293
 - literal numeric values, avoiding 292
 - naming things 291
 - premature optimization, avoiding 294
 - readability and correctness,
 - prioritizing 293, 294
 - scope of identifiers, minimizing 295, 296
 - unconditional jumps, avoiding 295
 - vertical spacing 293
- cold joint 238
- combinational logic 20
- comma-separated value (CSV) 231
- comma-separated values (CSV) 311

- communication security, MQTT project
 - reference link 286
- comparator 45
- Complementary Metal Oxide
 - Semiconductor (CMOS) 93
- comprehensive test coverage
 - code coverage, tracking 327, 328
 - ensuring 323
 - requirements traceability
 - matrix (RTM) 324-327
 - testing, need for 328, 329
- condition coverage 327
- Controller Area Network (CAN) 52, 53
- counting semaphore 72
- Cygwin
 - reference link 305

D

- data collection, examples
 - analog voltage measurement 319
 - digital signal capture 320
 - network packet capture 319
 - video recording 320
- deadlock 76, 77
- debugger 334
- debugging 329
- deserializer 275
- design entry 103, 142
- differential signals 93
- digital logic gates 17-19
- digital oscilloscope circuit board
 - 1 KHz calibration signal,
 - generating 265-269
 - ADC encoder clock, generating 265-269
 - ADC, testing 253-261
 - analog amplifiers, testing 250-252
 - basic functionality, checking 246, 247

- board power supplies, testing 247-249
- FPGA logic, adding 265
- I/O signals, checking 265
- modifications, making 261
- power supply 244, 245
- preparation, before applying power 244
- digital oscilloscope system
 - functional elements 272
 - overview 272-274
- digital signal processing
 - (DSP) operation 26
- Digital Signal Processors (DSPs) 49
- digital-to-analog converter (DAC) 339
- distributed RAM 91
- Domain Name System (DNS) 287
- dominant state 53
- double data rate (DDR) 275
- DSP slices 21
- Dynamic Host Configuration
 - Protocol (DCHP) 287
- dynamic RAM (DRAM) 94

E

- ÉCLAIR
 - reference link 298
- edge-sensitive device 19
- effective debugging techniques
 - about 329
 - binary search debugging
 - process, using 335, 336
 - dealing with, compilation errors
 - and warnings 330
 - dealing with, syntax 330
 - portions of system functionality,
 - disabling 336
 - problem, defining 332

- program issues, solving 336
- system functionality 333, 335
- test input, determining 333
- working with, static code analysis
 - and unit testing 331
- electrostatic discharge (ESD) 219
- Electrostatic Discharge (ESD) 190
- elements, embedded systems
 - about 4, 5
 - digital processing 6
 - firmware 7
 - input, from environment 8
 - memory 7
 - network communication 9, 10
 - output, to environment 9
 - power source 5
 - software 7
 - specialized circuitry 8
 - time base 6
- embedded system
 - hardware solution 46
 - software solution 46-48
- embedded systems
 - elements 4
 - FPGAs 16
 - operating, in real-time 11
 - sensor types, using 36
- embOS
 - about 83
 - reference link 83
- event-driven embedded systems 13
- event flags 73
- event groups 73
- existing code
 - working with 299, 300
- Extreme Programming (XP) 340

F

features, debugger

- breakpoint 334
- single-step code execution 334
- source-level debugging 334
- variable and memory display 334
- watchpoint 334

Field-Programmable Gate Arrays (FPGAs)

- in embedded systems 16

firmware 7

first-in first-out (FIFO) 280

first-in-first-out (FIFO) 92

flip flops 19-21

fluid flow sensors 38

flux

- functions 217
- no clean 218

force sensor 39

FPGA algorithm

- designing 272
- implementing 272

FPGA compilation process

- about 142
- constraints 160, 161
- design entry 142-147
- design optimization 149-153
- high-level synthesis 153-159
- logic synthesis 147-149
- optimization 160, 161

FPGA design

- AXI bus interface, adding 283, 284
- deserializer, adding 275-280
- FIFO buffer, adding 280, 282
- implementing 103
- MQTT protocol, adding 285-290

FPGA design, phases

- bitstream, generating 104

design entry 103

I/O planning 103

place and route process 104

synthesis 104

FPGA design project

- description 164, 165
- working with 164

FPGA development process

- about 100
- features, identifying 102
- functionality, allocating 102
- implementation, testing 105
- system requisites, defining 101

FPGA device

- block RAM 91, 92
- distributed RAM 91, 92
- features 92-94
- hardware processor cores 95
- I/O pins 92-94
- specialized hardware resources 94, 95
- using, in real-time embedded system designs 90, 91

FPGA implementation

- suitable algorithm types 162

FPGA I/O planning 103

FPGA languages

- block diagrams 98, 99
- C/C++ 99, 100
- implementing 95
- Verilog 97
- VHDL 95, 96

FPGA project

- bitstream, downloading to Arty A7 board 133, 134
- bitstream, implementing 130-133
- bitstream, programming to onboard flash memory 135-138

- bitstream, synthesizing 131-133
- creating 110-112
- developing 106
- implementing 106
- I/O signals, defining 127, 128
- logic behavior, testing 120-127
- top-level VHDL file, creating 129, 130
- VHDL source files, creating 112-119
- Vivado tools, installing 107-109
- FPGAs, elements
 - about 20, 22
 - Block RAM (BRAM) 21
 - DSP slices 21
 - flip flops 21
 - lookup tables 21
- FPGAs, embedded systems
 - about 16
 - digital logic gates 16-18
 - elements 20
 - flip flops 19, 20
 - hardware design languages 22-24
 - synthesis 22
 - using, benefits 25, 26
 - Xilinx FPGAs and development tools 26
- FPGA synthesis 22
- FreeRTOS
 - about 83
 - URL 83
- FreeRTOS_IP 287
- full adder 23
- full-duplex 51
- fume extractor 218
- functional testing
 - about 314
 - versus unit testing 313-315

G

- Gas Discharge Tube (GDT) 190
- gate array 16
- General-Purpose I/O (GPIO) 44-46
- Git
 - URL 304
 - version control 304
- GitHub
 - URL 304
- Global Navigation Satellite System (GNSS) 44
- Global Positioning System (GPS) 43, 44
- GPS system 44

H

- half adder 23
- half-duplex communication protocol 50
- hand soldering 220-234
 - repairs after reflow 234
- hard real-time systems 58
- hardware 7
- hardware compiler 142
- Hardware Description Language (HDL) 97
- Hardware Description Languages (HDLs) 147
- harmful ESD
 - risk, reducing 220
- heap 74
- heap fragmentation 75
- heap overflow 74
- high-performance embedded system development
 - best practices 337
 - code development 339, 340
 - designing, for test 337

- future growth requirements 338
- hardware design 338, 339
- strict version control,
 - maintaining 340, 341
- system-level testing 341, 342
- unit tests, developing as code 341

High-Speed Transceiver Logic (HSTL) 93

humidity sensors 38

Hysteresis 46

I

inertial sensor 43

INS system 44

Institute of Electrical and Electronics Engineers (IEEE) 96

INTEGRITY RTOS

- about 84
- URL 84

Intellectual Property (IP) 99

Inter-Integrated Circuit (I2C) 50

Internet of Things (IoT)

- about 10, 11
- devices and systems, examples 10, 11

Interrupt Service Routines (ISRs) 65

ionizing radiation 41

isopropyl alcohol (IPA) 236

K

KiCad

- about 178-180
- circuit components, connecting 182-190
- circuit components, placing 182-190
- component symbols, creating 190-195
- functions 178
- procedures 180, 181

kilobits (Kb) 91

L

LDRARules

- reference link 298

lead-free solder 218

lidar sensor 42

Light Detection and Ranging (LIDAR) 41

light-emitting diodes (LEDs) 335

light sensors 36

lightweight IP (lwIP) 287

liquidous temperature 228

Littelfuse CG7 Series data

- reference link 191

logic analyzers 320

Lookup Table (LUT) 143

lookup tables 21

Low Voltage CMOS (LVCMOS) 93

Low-Voltage Differential Signaling (LVDS) 94

Low Voltage Digital Signaling (LVDS) 49

Low Voltage TTL (LVTTL) 93

LWIP_DNS 287

M

magnetic field sensor 40

magnifier 215

master 51

master-in-servant-out (MISO) 51

master-out-servant-in (MOSI) 51

master-servant network 50

Memory Management Unit (MMU) 81

Memory Protection Unit (MPU) 81

message queue 72

Message Queuing and Telemetry Transport (MQTT)

- URL 273

MicroBlaze processor 98
 microcontroller 6
 microelectromechanical
 System (MEMS) 6
 microprocessor 6
 microscope 215, 216
 million samples per second (MSPS) 34
 modifications, making in digital
 oscilloscope circuit board
 components, adding 264
 components, removing 263, 264
 jumper wires, installing 262, 263
 PCB traces, cutting 261
 solder jumpers, installing 262, 263
 Motor Industry Software Reliability
 Association (MISRA)
 URL 298
 MQTT broker 285
 MQTT implementation,
 FreeRTOS distribution
 reference link 286
 multiply-accumulate (MAC) 94
 Multiply-Accumulate (MAC)
 operation 21
 mutual exclusion (mutex) 70, 71

N

Nanotechnology 40
 negative testing 315
 net 252
 netlist 104
 noise 34
 nominal conditions
 testing 312, 313

O

off-nominal conditions
 testing 312, 313
 open collector 50
 open drain 50
 operational amplifier (op amp) 251
 optical magnification 214-216
 OSH Park
 URL 210
 OSH Stencils
 URL 211
 oxidation 217

P

passive sensor 30, 31
 PC-Lint Plus
 reference link 298
 penetration testing 315
 periodic embedded systems 12, 13
 peripheral module (Pmod) 182
 photoresistor 36
 place process 104
 polarized capacitors 188
 post-assembly board
 cleaning 236
 electrical short checking 238, 239
 flux removal 236
 inspection 236
 visual inspection 237
 potentiometer 45
 preemptive multitasking 64
 pressure sensors 37
 Printed Circuit Board (PCB)
 laying out 200-209
 priority inheritance 80
 priority inversion 77-80

- processor utilization 69
- project schematic diagram
 - annotation, performing 199
 - developing 195, 196
 - differential signal pairs, creating 198
 - electrical rules checking,
 - performing 199
 - global labels, adding 197
 - offboard connections, creating 198, 199
 - signal labels, adding 197
 - text annotations, adding 196
- publish-subscribe communication
 - system 285

Q

- QNX Neutrino RTOS
 - about 84
 - reference link 84
- queues 72

R

- Radio Detection and Ranging (RADAR) 41
- random access memory (RAM) 6
- rate-monotonic Scheduling (RMS) 69
- read-only memory (ROM) 6
- real-time 58, 59
- real-time embedded system
 - attributes 59, 60
 - multiple tasks, performing 60-68
 - rate-monotonic Scheduling (RMS) 69, 70
 - states 65
- real-time embedded system designs
 - FPGA device, using 90, 91
- real-time operating system (RTOS) 14, 15

- Real-Time Operating Systems (RTOSes) 57
 - about 81
 - deadlock 76, 77
 - dynamic memory allocation 74
 - embOS 83
 - event flags 73
 - features, and challenges 70
 - FreeRTOS 83
 - INTEGRITY RTOS 84
 - mutexes 70, 71
 - non-technical attributes 82
 - priority inversion 77-80
 - QNX Neutrino RTOS 84
 - queues 72
 - semaphores 72
 - technical attributes 81
 - timers 73
 - VxWorks 85
 - μC/OS-III 84, 85
- recessive state 53
- reflow soldering 223, 227-229, 233
- reflow soldering temperature profile
 - cooldown 228
 - preheat 227
 - reflow 228
 - soak 227
- repeatable test results
 - achieving 316, 317
- requirements-driven testing 309-312
- requirements traceability matrix (RTM) 324-327
- rework station 221
- ripple-carry adder 143
- Rosin 219
- route process 104
- RSM
 - reference link 298

RTOS, dynamic memory allocation

 heap fragmentation 75

 memory leaks 74

Ruby

 reference link 305

S

sample-and-hold circuit 33

scheduling 65

semaphores 72

sensitivity list 145

sensor 30

 Analog signals 48, 49

 communicating with 44

 Controller Area Network (CAN) 52, 53

 General-Purpose I/O (GPIO) 44-46

 Inter-Integrated Circuit (I2C) 50

 Serial Peripheral Interface (SPI) 51, 52

 wireless technologies 54

sensor circuits

 applying 32-36

sensor data

 processing 55

sensors

 fluid flow sensors 38

sensor types

 audio sensors 39

 chemical sensors 40

 force sensor 39

 GPS 43, 44

 humidity sensors 38

 inertial sensor 43

 ionizing radiation 41

 Light Detection and Ranging

 (LIDAR) 41

 light sensors 36

 magnetic field sensor 40

 pressure sensors 37

 Radio Detection and Ranging

 (RADAR) 41

 temperature sensors 37

 ultrasonic 39

 using in embedded systems 36

 video and infrared 42

sequential logic 21

serial clock (SCL) 50

serial clock (SCLK) 51

serial data (SDA) 50

Serial Peripheral Interface

 (SPI) 51, 52, 275

servants 51

severe messages 301

silicon compiler 142

simulated environment

 testing in 316

single-ended signal 93

smart sensor 30, 31

smoke-test 245

soft processor 16

soft real-time systems 58

software 7

solder 218, 219

solder bridges 216

soldering

 issues 237

 safety tips 229

solder jumper 262

solder paste

 applying, to stencil 225, 226

solder paste application 223, 225

solder wick 222

source code analyzer

 messages 302, 303

source code version control 304

statement coverage 327

- static code analysis 297
 - using, effectively 299
- static code analysis tools
 - about 298
 - Clang-Tidy 298
 - ECLAIR 298
 - LDRArules 298
 - PC-Lint Plus 298
 - RSM 298
- static RAM (SRAM) 104
- static source code analyzer 297
- Stub-Series Terminated Logic (SSTL) 93
- suitable algorithm types, FPGA
 - implementation
 - about 162
 - high-speed data streams
 - processing algorithms 162
 - nonstandard data sizes, using
 - for algorithms 163
 - parallel algorithms 163
- surface-mount technology (SMT) 179
- synchronous sequential logic 148
- synthesis 104
- synthesizable language subsets 97
- system-level tests
 - designing 308, 309
- system-level tests, approaches
 - negative testing 315
 - nominal and off-nominal
 - conditions, testing 312, 313
 - penetration testing 315
 - repeatable test results,
 - achieving 316, 317
 - requirements-driven testing 309-312
 - simulated environment, testing in 316
 - test plan, developing 317, 319
 - unit testing, versus functional
 - testing 313-315

- system on a chip (SoC) 6
- system requirements, analysis
 - clarity 310
 - completeness 310
 - testability 310
- system requirements,
 - fundamental methods
 - analysis 312
 - demonstration 312
 - inspection 312
 - test 312

T

- tack 226
- task 15
- Task Control Block (TCB) 65
- test-driven development (TDD)
 - about 304
 - applying, to embedded systems 305
- tests results
 - conducting 319
 - data collection, identifying 319, 320
 - quick-look assessment 321
 - recording 319
 - regression testing, on tested
 - code 322, 323
 - repeating 322
 - system under test, configuring 320
 - test procedures, executing 320, 321
- thermistor 30, 37
- thermocouple 37
- thread 15
- through-hole components
 - installing 235
- through-hole technology (THT) 179
- timer callback function 73
- timers 73

tinny 222
tombstoning 238
topic 285
traces 178
Transistor-Transistor Logic (TTL) 93
Transition-Minimized Differential
 Signaling (TMDS) 267
tweezers 216

U

ultrasonic 39
unit testing
 about 313
 versus functional testing 313-315
Universal Serial Bus (USB) 315
unpolarized capacitors 188

V

Verilog 23, 97
version control
 with Git 304
VHDL 23
VHSIC Hardware Description
 Language (VHDL) 95, 96
vias 179
video and infrared 42
Video sensors 36
VxWorks
 about 85
 reference link 85

W

wall clock time 6
wetting 217
white box testing 314
wireless technologies 54
 Bluetooth 54
 cellular network 55
 radio frequency ID (RFID) 54
 Wi-Fi 54
Wireshark
 URL 319
Worst Negative Slack (WNS) 151

X

XADC 34
Xilinx Software Command-line
 Tool (XSCT) 274